pyimagesearch
be awesome at building image search engines

OpenCV and Python K-Means Color Clustering

by **Adrian Rosebrock** on May 26, 2014 in **Image Processing**, **Tutorials**

Take a second to look at the *Jurassic Park* movie poster above.

What are the *dominant* colors? (i.e. the colors that are represented most in the image)

Well, we see that the background is largely **black**. There is some **red** around the T-Rex. And there is some **yellow** surrounding the actual logo.

It's pretty simple for the human mind to pick out these colors.

But what if we wanted to create an algorithm to *automatically* pull out these colors?

Looking for the source code to this post?
Jump right to the downloads section.

You might think that a color histogram is your best bet…

But there's actually a more interesting algorithm we can apply — k-means clustering.

In this blog post I'll show you how to use OpenCV, Python, and the k-means clustering algorithm to find the most dominant colors in an image.

OpenCV and Python versions:
This example will run on **Python 2.7/Python 3.4+** and **OpenCV 2.4.X/OpenCV 3.0+**.

K-Means Clustering

So what exactly is k-means?

K-means is a clustering algorithm.

The goal is to partition *n* data points into *k* clusters. Ea
with the nearest mean. The mean of each cluster is ca

Overall, applying k-means yields *k* separate clusters
particular cluster are considered to be "more similar" t
clusters.

In our case, we will be clustering the pixel intensities c
have *MxN* pixels, each consisting of three component

We will treat these *MxN* pixels as our data points and

Pixels that belong to a given cluster will be more simil

er.

One caveat of k-means is that we need to specify the number of clusters we want to generate ***ahead of time***. There are algorithms that automatically select the optimal value of *k*, but these algorithms are outside the scope of this post.

OpenCV and Python K-Means Color Clustering

Alright, let's get our hands dirty and cluster pixel intensities using OpenCV, Python, and k-means:

OpenCV and Python K-Means Color Clustering — Python

```python
# import the necessary packages
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt
import argparse
import utils
import cv2

# construct the argument parser and parse th
ap = argparse.ArgumentParser()
ap.add_argument("-i", "--image", required =
```

```
11 ap.add_argument("-c", "--clusters", required = True, type = int,
12     help = "# of clusters")
13 args = vars(ap.parse_args())
14
15 # load the image and convert it from BGR to RGB so that
16 # we can dispaly it with matplotlib
17 image = cv2.imread(args["image"])
18 image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
19
20 # show our image
21 plt.figure()
22 plt.axis("off")
23 plt.imshow(image)
```

Lines 2-6 handle importing the packages we need. We'll use the scikit-learn implementation of k-means to make our lives easier — no need to re-implement the wheel, so to speak. We'll also be using `matplotlib` to display our images and most dominan
use `argparse`. The `utils` package contains two help
`cv2` package contains our Python bindings to the Ope

Lines 9-13 parses our command line arguments. We
path to where our image resides on disk, and `--clus`
generate.

On Lines 17-18 we load our image off of disk and the
Remember, OpenCV represents images as multi-dime
stored in BGR order rather than RGB. To remedy this

Finally, we display our image to our screen using `mat`

As I mentioned earlier in this post, our goal is to gener
our *MxN* image as our data points.

In order to do this, we need to re-shape our image to b

OpenCV and Python K-Means Color Clustering — Python

```
25 # reshape the image to be a list of pixels
26 image = image.reshape((image.shape[0] * image.shape[1], 3))
```

This code should be pretty self-explanatory. We are simply re-shaping our NumPy array to be a list of RGB pixels.

The 2 lines of code:

Now that are data points are prepared, we can write these **2 lines of code** using k-means to find the most dominant colors in an image:

OpenCV and Python K-Means Color Clustering — Python

```
28 # cluster the pixel intensities
29 clt = KMeans(n_clusters = args["clusters"])
30 clt.fit(image)
```

We are using the scikit-learn implementation of k-means to avoid re-implementing the algorithm. There is also a k-means built into OpenCV, but if you have ever done any type of machine learning in Python before (or if you ever intend to), I suggest using the scikit-learn package.

We instantiate `KMeans` on **Line 29**, supplying the number of clusters we wish to generate. A call to `fit()` method on **Line 30** clusters our list of pixels.

That's all there is to clustering our RGB pixels using Python and k-means.

Scikit-learn takes care of everything for us.

However, in order to display the most dominant colors

Let's open up a new file, `utils.py`, and define the ce

```
OpenCV and Python K-Means Color Clustering
 1  # import the necessary packages
 2  import numpy as np
 3  import cv2
 4
 5  def centroid_histogram(clt):
 6      # grab the number of different clusters
 7      # based on the number of pixels assigned
 8      numLabels = np.arange(0, len(np.unique(c
 9      (hist, _) = np.histogram(clt.labels_, bi
10
11      # normalize the histogram, such that it
12      hist = hist.astype("float")
13      hist /= hist.sum()
14
15      # return the histogram
16      return hist
```


As you can see, this method takes a single parameter, `clt`. This is our k-means clustering object that we created in `color_kmeans.py`.

The k-means algorithm assigns each pixel in our image to the closest cluster. We grab the number of clusters on **Line 8** and then create a histogram of the number of pixels assigned to each cluster on **Line 9**.

Finally, we normalize the histogram such that it sums to one and return it to the caller on **Lines 12-16**.

In essence, ***all this function is doing is counting the number of pixels that belong to each cluster.***

Now for our second helper function, `plot_colors`:


```
OpenCV and Python K-Means Color Clustering
```

```
18 def plot_colors(hist, centroids):
19     # initialize the bar chart representing the relative frequency
20     # of each of the colors
21     bar = np.zeros((50, 300, 3), dtype = "uint8")
22     startX = 0
23
24     # loop over the percentage of each cluster and the color of
25     # each cluster
26     for (percent, color) in zip(hist, centroids):
27         # plot the relative percentage of each cluster
28         endX = startX + (percent * 300)
29         cv2.rectangle(bar, (int(startX), 0), (int(endX), 50),
30             color.astype("uint8").tolist(), -1)
31         startX = endX
32
33     # return the bar chart
34     return bar
```

The `plot_colors` function requires two parameters: `hist`, which is the `centroid_histogram` function, and `centroids`, which are the clusters generated by the k-means algorithm.

On **Line 21** we define a *300×50* pixel rectangle to hold

We start looping over the color and percentage contrib current color contributes to the image on **Line 29**. We **Line 34**.

Again, this function performs a very simple task — ge assigned to each cluster based on the output of the c

Now that we have our two helper functions defined, w

OpenCV and Python K-Means Color Clustering

```
32 # build a histogram of clusters and then cre
33 # representing the number of pixels labeled
34 hist = utils.centroid_histogram(clt)
35 bar = utils.plot_colors(hist, clt.cluster_centers_)
36
37 # show our color bart
38 plt.figure()
39 plt.axis("off")
40 plt.imshow(bar)
41 plt.show()
```

On **Line 34** we count the number of pixels that are assigned to each cluster. And then on **Line 35** we generate the figure that visualizes the number of pixels assigned to each cluster.

Lines 38-41 then displays our figure.

To execute our script, issue the following command:

OpenCV and Python K-Means Color Clustering

```
1 $ python color_kmeans.py --image images/jp.pn
```


If all goes well, you should see something similar to be

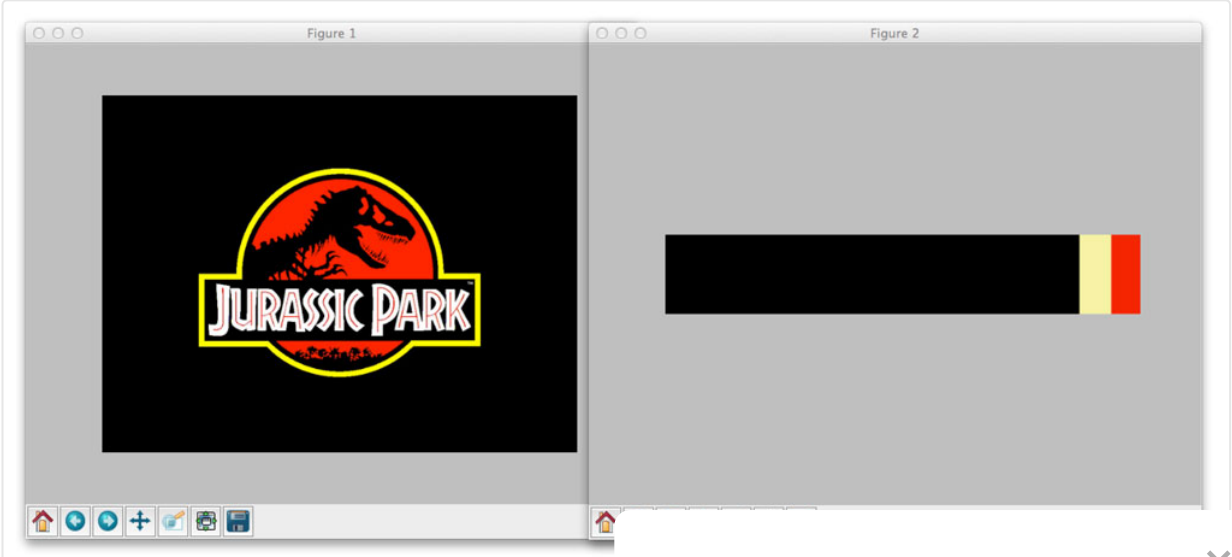

Figure 1: Using Python, OpenCV, and k-means

Here you can see that our script generated three clus
command line argument). The most dominant clusters
represented in the *Jurassic Park* movie poster.

Let's apply this to a screenshot of *The Matrix*:

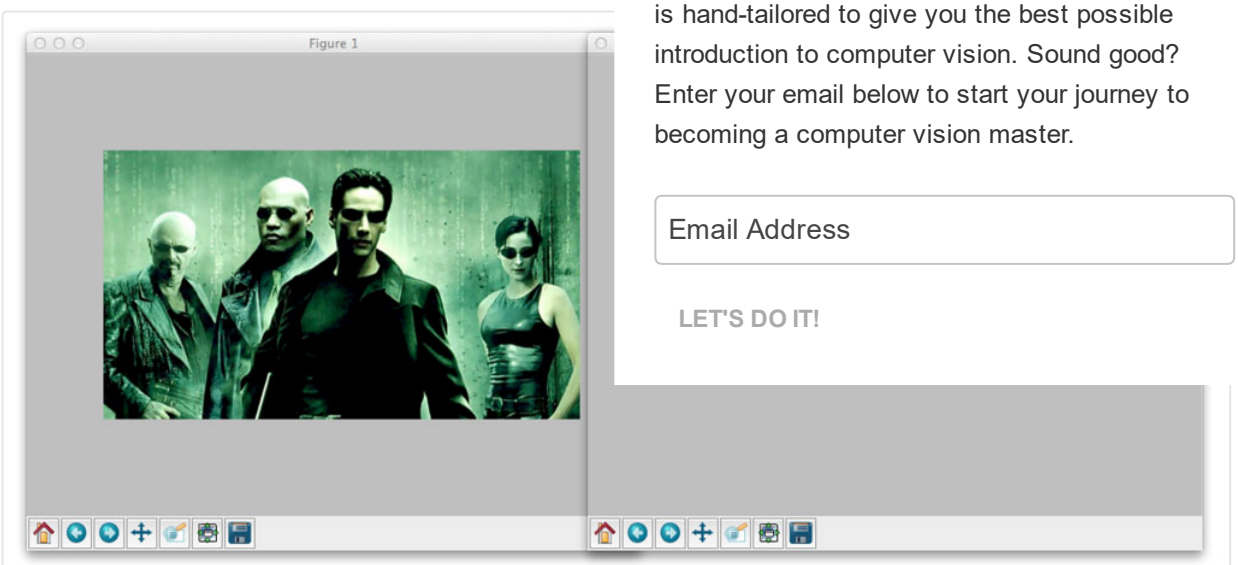

Figure 2: Finding the four most dominant colors using k-means in our *The Matrix* image.

This time we told k-means to generate four clusters. As you can see, black and various shades of green are the most dominant colors in the image.

Finally, let's generate five color clusters for this Batman image:

Free 21-day crash course on computer vision & image search engines

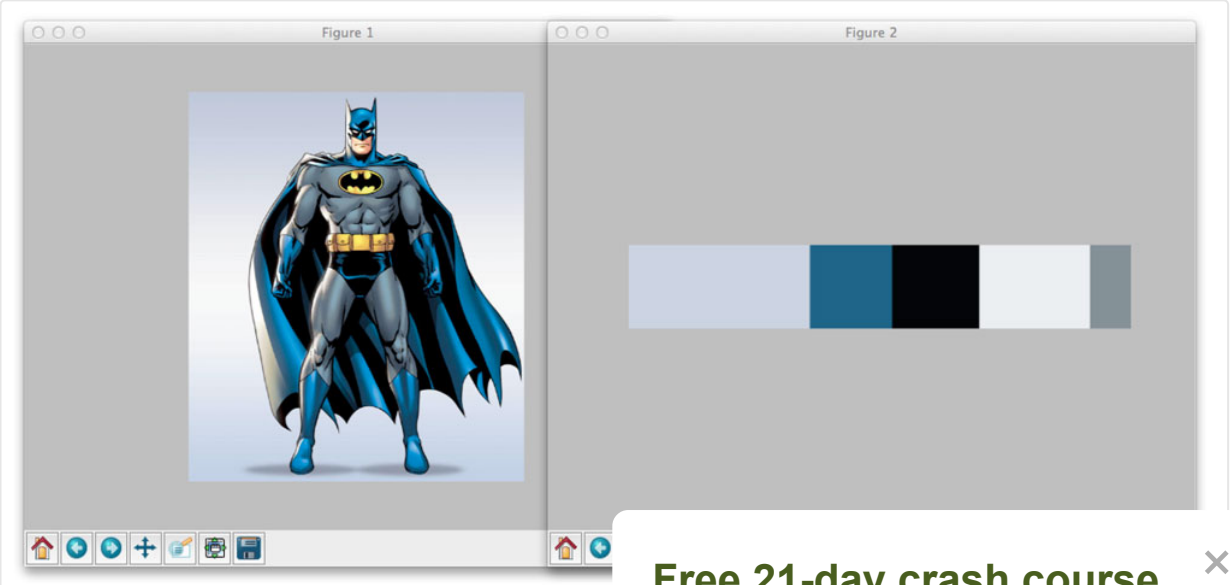

Figure 3: Applying OpenCV and k-means clustering

So there you have it.

Using OpenCV, Python, and k-means to cluster RGB the image is actually quite simple. Scikit-learn takes c this post was used to glue all the pieces together.

Summary

In this blog post I showed you how to use OpenCV, P in the image.

K-means is a clustering algorithm that generates *k* clu *k* must be specified ahead of time. Although algorithm outside the scope of this blog post.

In order to find the most dominant colors in our image, we treated our pixels as the data points and then applied k-means to cluster them.

We used the scikit-learn implementation of k-means to avoid having to re-implement it.

I encourage you to apply k-means clustering to our own images. In general, you'll find that smaller number of clusters (*k* <= *5*) will give the best results.

Downloads:

If you would like to download the code and images used in this post, please enter your email address in the form below. Not only will you get a .zip of the code, I'll Computer Vision and Image Search Engines, includin Sound good? If so, enter your email address and I'll s