Using K-means for color compression

In [1]:
```
import numpy as np
import pandas as pd

%pylab inline
import plotly.graph_objects as go

from sklearn.cluster import KMeans
```

```
%pylab is deprecated, use %matplotlib inline and import the required librarie
s.
Populating the interactive namespace from numpy and matplotlib
```

Read in data The "data" in this case is not a pandas dataframe. It's a photograph, which we'll convert into a numerical array.

In [2]:
```
# Read in a photo
img = plt.imread('images_(79)[1].jpeg')
```

In [3]:
```
1 img
```

Out[3]:
```
array([[[220, 189, 125],
        [220, 186, 122],
        [220, 185, 117],
        ...,
        [206, 135, 133],
        [201, 158, 152],
        [191, 169, 158]],

       [[221, 187, 124],
        [219, 185, 121],
        [218, 183, 115],
        ...,
        [211, 131, 130],
        [205, 152, 146],
        [197, 164, 155]],

       [[217, 181, 121],
        [218, 180, 117],
        [217, 180, 112],
        ...,
        [219, 125, 125],
        [211, 142, 137],
        [206, 156, 149]],

       ...,

       [[  1,   1,   1],
        [255, 255, 255],
        [255, 255, 255],
        ...,
        [255, 255, 255],
        [255, 255, 255],
        [255, 255, 255]],

       [[  1,   1,   1],
        [255, 255, 255],
        [255, 255, 255],
        ...,
        [255, 255, 255],
        [255, 255, 255],
        [255, 255, 255]],

       [[  1,   1,   1],
        [255, 255, 255],
        [255, 255, 255],
        ...,
        [255, 255, 255],
        [255, 255, 255],
        [255, 255, 255]]], dtype=uint8)
```

In [4]:
```
# Display the photo and its shape
print(img.shape)
plt.imshow(img)
plt.axis('off');
```

```
(280, 463, 3)
```

In [5]:
```
# Reshape the image so that each row represents a single pixel
# defined by three values: R, G, B
img_flat = img.reshape(img.shape[0]*img.shape[1], 3)
img_flat[:5, :]
```

Out[5]:
```
array([[220, 189, 125],
       [220, 186, 122],
       [220, 185, 117],
       [225, 189, 115],
       [230, 193, 113]], dtype=uint8)
```

In [6]:
```
# Create a pandas df with r, g, and b as columns
img_flat_df = pd.DataFrame(img_flat, columns = ['r', 'g', 'b'])
img_flat_df.head()
```

Out[6]:

	r	g	b
0	220	189	125
1	220	186	122
2	220	185	117
3	225	189	115
4	230	193	113

In [7]:

```
# Create 3D plot where each pixel in the `img` is displayed in its actual
trace = go.Scatter3d(x = img_flat_df.r,
                     y = img_flat_df.g,
                     z = img_flat_df.b,
                     mode='markers',
                     marker=dict(size=1,
                                 color=['rgb({},{},{})'.format(r,g,b) for
                                        in zip(img_flat_df.r.values,
                                               img_flat_df.g.values,
                                               img_flat_df.b.values)],
                                 opacity=0.5))

data = [trace]

layout = go.Layout(margin=dict(l=0,
                               r=0,
                               b=0,
                               t=0),
                               )

fig = go.Figure(data=data, layout=layout)
fig.update_layout(scene = dict(
                    xaxis_title='R',
                    yaxis_title='G',
                    zaxis_title='B'),
                  )
fig.show()
```

☹

Cluster the data: k = 1

In [8]:
```
# Instantiate the model
kmeans = KMeans(n_clusters=1, random_state=42).fit(img_flat)
```

In [9]:
```
# Copy `img_flat` so we can modify it
img_flat1 = img_flat.copy()

# Replace each row in the original image with its closest cluster center
for i in np.unique(kmeans.labels_):
    img_flat1[kmeans.labels_==i,:] = kmeans.cluster_centers_[i]

# Reshape the data back to (640, 480, 3)
img1 = img_flat1.reshape(img.shape)

plt.imshow(img1)
plt.axis('off');
```

So what happened? Well, let's run through the K-means steps:

We randomly placed our centroid in the colorspace. We assigned each point to its nearest centroid. Since there was only one centroid, all points were assigned to the same centroid, and thus to the same cluster. We updated the centroid's location to the mean location of all of its points. Again, since there is only a single centroid, it updated to the mean location of every point in the image. Repeat until the model converges. In this case, it only took one iteration for the model to converge. We then updated each pixel's RGB values to be the same as the centroid's. The result is the image of our tulips when every pixel is replaced with the average color. The average color of this photo was brown—all the colors muddled together.

We can verify this for ourselves by manually calculating the average for each column in the flattened array. This will give us the average R value, G value, and B value.

In [10]:
```
# Calculate mean of each column in the flattened array
column_means = img_flat.mean(axis=0)

print('column means: ', column_means)
```

```
column means:  [179.55814563 111.90674946  79.88253625]
```

Now, we can compare this to what the K-means model calculated as the final location of its one centroid.

In [11]:
```
1 print('cluster centers: ', kmeans.cluster_centers_)
```

```
cluster centers:  [[179.55814563 111.90674946  79.88253625]]
```

They're the same! Now, let's return to the 3-D rendering of our data, only this time we'll add the centroid.

In [12]:

```
# Create 3-D plot where each pixel in the `img` is displayed in its actual
trace = go.Scatter3d(x = img_flat_df.r,
                     y = img_flat_df.g,
                     z = img_flat_df.b,
                     mode='markers',
                     marker=dict(size=1,
                                 color=['rgb({},{},{})'.format(r,g,b) for
                                        r,g,b in zip(img_flat_df.r.values,
                                                     img_flat_df.g.values,
                                                     img_flat_df.b.values)
                                 opacity=0.5))

data = [trace]

layout = go.Layout(margin=dict(l=0,
                               r=0,
                               b=0,
                               t=0))

fig = go.Figure(data=data, layout=layout)


# Add centroid to chart
centroid = kmeans.cluster_centers_[0].tolist()

fig.add_trace(
    go.Scatter3d(x = [centroid[0]],
                 y = [centroid[1]],
                 z = [centroid[2]],
                 mode='markers',
                 marker=dict(size=7,
                             color=['rgb(125.79706706,78.8178776,42.580901
                             opacity=1))
)
fig.update_layout(scene = dict(
                    xaxis_title='R',
                    yaxis_title='G',
                    zaxis_title='B'),
                  )
fig.show()
```

☹

We can see the centroid as a large circle in the middle of the colorspace. (If you can't, just click on the image and spin/zoom it.) Notice that this is the "center of gravity" of all the points in the graph.

Now let's try something else. Let's refit a K-means model to the data, this time using k = 3. Take a moment to consider what you might expect to result from this. Go through the steps of what the model is doing like we did above. What colors are you likely to see?

Cluster the data: *k* = 3

In [13]:
```
# Instantiate k-means model for 3 clusters
kmeans3 = KMeans(n_clusters=3, random_state=42).fit(img_flat)

# Check the unique values of what's returned by the .labels_ attribute
np.unique(kmeans3.labels_)
```

Out[13]:
```
array([0, 1, 2])
```

The .cluster_centers_ attribute returns an array where each element represents the coordinates of a centroid (i.e., their RGB values). We'll use these coordinates as we did previously to generate the colors that are represented by our centroids.

In [14]:
```
# Assign centroid coordinates to `centers` variable
centers = kmeans3.cluster_centers_
centers
```

Out[14]:
```
array([[210.43027568, 103.15660618,  65.82105407],
       [238.60707255, 224.70164054, 177.11527525],
       [ 79.27359372,  42.74099099,  32.04084267]])
```

Now we'll create a helper function to easily display RGB values as color swatches, and use it to check the colors of the model's centroids.

In [15]:
```
# Helper function that creates color swatches
def show_swatch(RGB_value):
    '''
    Takes in an RGB value and outputs a color swatch
    '''
    R, G, B = RGB_value
    rgb = [[np.array([R,G,B]).astype('uint8')]]
    plt.figure()
    plt.imshow(rgb)
    plt.axis('off');
```

In [16]:
```
# Display the color swatches
for pixel in centers:
    show_swatch(pixel)
```

Hopefully, you hypothesized that we'd see similar colors as a result of a 3-cluster model. If you examine the original image of the tulips, it's apparent that there are generally three dominant colors: reds, greens, and golds/yellows, which is very close to what the model returned.

Just as before, let's now replace each pixel in the original image with the RGB value of the centroid to which it was assigned.

In [17]:
```
# Helper function to display our photograph when clustered into k clusters
def cluster_image(k, img=img):
    '''
    Fits a K-means model to a photograph.
    Replaces photo's pixels with RGB values of model's centroids.
    Displays the updated image.

    Args:
      k:    (int)          - Your selected K-value
      img:  (numpy array)  - Your original image converted to a numpy arra

    Returns:
      The output of plt.imshow(new_img), where new_img is a new numpy arra
      where each row of the original array has been replaced with the \
      coordinates of its nearest centroid.
    '''

    img_flat = img.reshape(img.shape[0]*img.shape[1], 3)
    kmeans = KMeans(n_clusters = k, random_state = 42).fit(img_flat)
    new_img = img_flat.copy()

    for i in np.unique(kmeans.labels_):
        new_img[kmeans.labels_ == i, :] = kmeans.cluster_centers_[i]

    new_img = new_img.reshape(img.shape)

    return plt.imshow(new_img), plt.axis('off');

```

Generate image when k=3

cluster_image(3);

We now have a photo with just three colors. Each pixel's RGB values correspond to the values of its nearest centroid.

We can return once more to our 3-D colorspace. This time, we'll re-color each dot in the colorspace to correspond with the color of its centroid. This will allow us to see how the K-means algorithm clustered our data spatially.

Again, don't concern yourself so much with the code. Feel free to skip down to the graph.

In [18]:
```
# Just to get an understanding of what the data structures look like

print(kmeans3.labels_.shape)
print(kmeans3.labels_)
print(np.unique(kmeans3.labels_))
print(kmeans3.cluster_centers_)
```

```
(129640,)
[1 1 1 ... 1 1 1]
[0 1 2]
[[210.43027568 103.15660618  65.82105407]
 [238.60707255 224.70164054 177.11527525]
 [ 79.27359372  42.74099099  32.04084267]]
```

In [19]:
```
# Create a new column in the df that indicates the cluster number of each
# (as assigned by Kmeans for k=3)
img_flat_df['cluster'] = kmeans3.labels_
img_flat_df.head()
```

Out[19]:

	r	g	b	cluster
0	220	189	125	1
1	220	186	122	1
2	220	185	117	1
3	225	189	115	1
4	230	193	113	1

In [20]:
```
# Create helper dictionary to map RGB color values to each observation in
series_conversion = {0: 'rgb' +str(tuple(kmeans3.cluster_centers_[0])),
                     1: 'rgb' +str(tuple(kmeans3.cluster_centers_[1])),
                     2: 'rgb' +str(tuple(kmeans3.cluster_centers_[2])),
                     }
series_conversion
```

Out[20]:
```
{0: 'rgb(210.43027567550294, 103.1566061821888, 65.82105407130598)',
 1: 'rgb(238.60707254830987, 224.7016405395384, 177.11527524605327)',
 2: 'rgb(79.27359371564982, 42.74099099098083, 32.04084267194157)'}
```

In [21]:
```
# Replace the cluster numbers in the 'cluster' col with formatted RGB valu
# (made ready for plotting)
img_flat_df['cluster'] = img_flat_df['cluster'].map(series_conversion)
img_flat_df.head()
```

Out[21]:

	r	g	b	cluster
0	220	189	125	rgb(238.60707254830987, 224.7016405395384, 177...
1	220	186	122	rgb(238.60707254830987, 224.7016405395384, 177...
2	220	185	117	rgb(238.60707254830987, 224.7016405395384, 177...
3	225	189	115	rgb(238.60707254830987, 224.7016405395384, 177...
4	230	193	113	rgb(238.60707254830987, 224.7016405395384, 177...

In [22]:
```
# Replot the data, now showing which cluster (i.e., color) it was assigned

trace = go.Scatter3d(x = img_flat_df.r,
                     y = img_flat_df.g,
                     z = img_flat_df.b,
                     mode='markers',
                     marker=dict(size=1,
                                 color=img_flat_df.cluster,
                                 opacity=1))

data = trace

layout = go.Layout(margin=dict(l=0,
                               r=0,
                               b=0,
                               t=0))

fig = go.Figure(data=data, layout=layout)
fig.show()
```

☹

You may be thinking to yourself that you would have clustered the data differently based on the distribution of points that you saw in the first 3-D plot. For example, why is there a sharp line that separates red and green, when there doesn't appear to be any empty space there in the data?

You're not incorrect. Even though there's no such thing as "wrong" clustering, some ways can definitely be better than others.

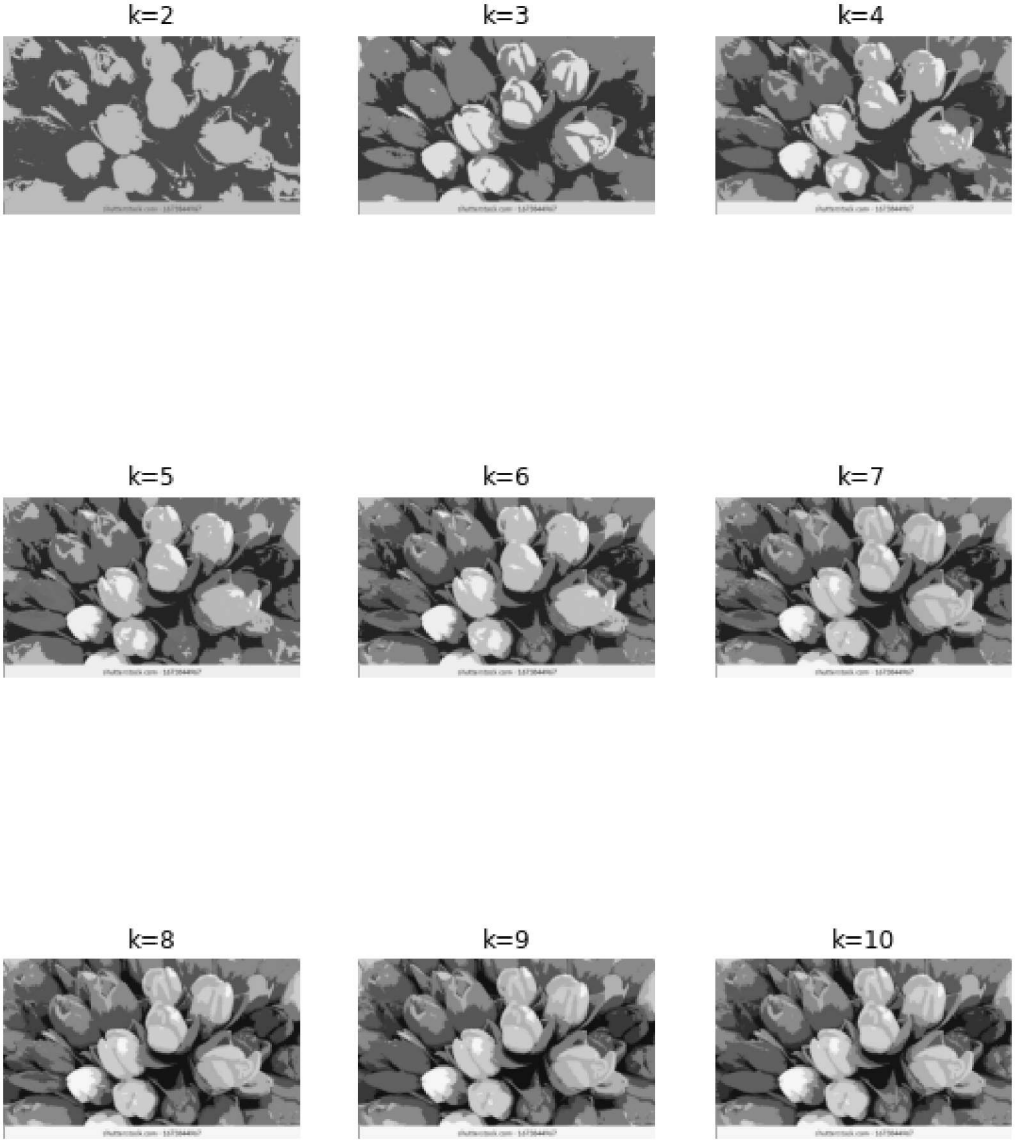
You'll notice in the original 3-D rendering that there are long bands—not round balls—of clustered data. K-means works best when the clusters are more circular, because it tries to minimize distance from point to centroid. It may be worth trying a different clustering algorithm if you want to cluster a long, narrow, continuous band of data. (More on these later!)

Nonetheless, K-means successfully compresses the colors of this photograph. This process can be applied for any value of k. Here's the output of each photo for k = 2–10.

Cluster the data: *k* = 2-10

In [23]:

```
# Helper function to
def cluster_image_grid(k, ax, img=img):
    '''
    Fits a K-means model to a photograph.
    Replaces photo's pixels with RGB values of model's centroids.
    Displays the updated image on an axis of a figure.

    Args:
      k:    (int)          - Your selected K-value
      ax:   (int)          - Index of the axis of the figure to plot to
      img:  (numpy array)  - Your original image converted to a numpy arra

    Returns:
      A new image where each row of the ori array has been replaced with t
      coordinates of its nearest centroid.
    '''
    img_flat = img.reshape(img.shape[0]*img.shape[1], 3)
    kmeans = KMeans(n_clusters=k, random_state=42).fit(img_flat)
    new_img = img_flat.copy()

    for i in np.unique(kmeans.labels_):
        new_img[kmeans.labels_==i, :] = kmeans.cluster_centers_[i]

    new_img = new_img.reshape(img.shape)
    ax.imshow(new_img)
    ax.axis('off')

fig, axs = plt.subplots(3, 3)
fig = matplotlib.pyplot.gcf()
fig.set_size_inches(9, 12)
axs = axs.flatten()
k_values = np.arange(2, 11)
for i, k in enumerate(k_values):
    cluster_image_grid(k, axs[i], img=img)
    axs[i].title.set_text('k=' + str(k))
```



Notice that it becomes increasingly difficult to see the difference between the images each time a color is added. This is a visual example of something that happens with all clustering models, even if the data is not an image that you can see. As you group the data into more and more clusters, additional clusters beyond a certain point contribute less and less to your understanding of your data.

In [ ]:
```
1
```