Assignment 4

Part I: K-means Segmentation ¶

Problem 1 (soft-max)

Following the material on slide 54 in topic 9, derive the optimal "soft" clustering (distribution over clusters) at point p

$$\mathbf{S}_p = \{S_p^k \mid 1 \leq k \leq K,\ S_p^k \geq 0,\ \sum_k S_P^k = 1\}$$

that Lloyd's algorithm would obtain when re-estimating segmentation for fixed cluster models parameters μ_k. Note that the total K-means objective $E(\mathbf{S}, \mu)$ as a function of segmentation $\mathbf{S}$ (when μ is fixed) is a sum of independent terms for every pixel. When computing optimal distribution $\mathbf{S}_p$ it is enough to focus on the terms dependent only on its components S_p^k. For the K-means formulation on slide 54, these terms are

$$-\sum_{k=1}^{K} S_p^k a_p^k \ - \ T\ H(\mathbf{S}_p) \qquad (*)$$

where constant $a_p^k := \log P(f_p|\mu_k)$ (assuming fixed μ) corresponds to k-th cluster's log-likelihood at the observed feature point f_p, constant T represents a so-called "temperature" parameter, and $H(\mathbf{S}_p) := -\sum_k S_p^k \log S_p^k$ is the entropy of distribution $\mathbf{S}_p$.

Use your solution to show what happens with the optimal distribution S_p when the temperature parameter reduces to zero $T \longrightarrow 0$.

HINT 1: Since optimization over $\mathbf{S}_p$ must be done over probability distributions, you should use Lagrangian formulation that combines the constraint $\sum_k S_p^k = 1$ with $(*)$ into the *Lagrangian*:

$$L(\mathbf{S}_p, \lambda) \ = \ -\sum_{k=1}^{K} S_p^k a_p^k \ - \ T\ H(\mathbf{S}_p) \ + \ \lambda \left(\sum_{k=1}^{K} S_p^k - 1 \right) \qquad (**)$$

where Lagrange multiplier λ is an additional optimization variable.

HINT 2: Similarly to optimization in single-variate functions, you can find extrema points for the multi-variate Lagrangian $(**)$ by finding values of variables $(S_p^1, \ldots, S_p^K, \lambda)$ where its derivative (gradient) equals zero. That is, the whole problem boils down to solving the system of $K+1$ equations $\nabla L = \mathbf{0}$ for the Lagrangian in $(**)$.

HINT 3: The goal of this excercise is to see how adding the entropy affects a linear loss. Optimization of $(*)$ over distributions $\mathbf{S}_p$ should result in the, so-called, soft-max operator applicable to arbitrary K potentials

$\{a_p^k \mid 1 \leq k \leq K\}$.

Problem 2 (Mahalanobis distance, decorrelation, etc.)

Let $X \in R^N$ be a Gaussian random vector with given mean μ and covariance matrix Σ. Find $N \times N$ matrix A such that linear transformation $Y = AX$ gives a random vector Y with covariance $\Sigma_Y = \mathbf{I}$. That is, the components of the transformed random vector Y should be i.i.d. You should derive an equation for matrix A assuming as given eigen-decomposition of the covariance matrix $\Sigma = U\Lambda U^T$ where $\Lambda = diag(s_1, \ldots, s_n)$ is a diagonal matrix of (non-negative!) eigen-values and U is an orthogonal $N \times N$ matrix (its columns are unit eigen-vectors of Σ).

HINT: you should solve the following (equivalent) simple geometric problem on "linear warps" (linear domain transforms): find a linear transformation A of points in R^N such that Mahalanobis distances (see slide 57) between any two given vectors $X, \mu \in R^N$ are equivalent to Euclidean distances between the corresponding vectors $Y = AX$ and $m = A\mu$ in the transformed space, that is,

$$\|X - \mu\|_\Sigma^2 \quad = \quad \|Y - m\|^2.$$

The proof should be simple if you use linear algebraic expressions for two squared metrics above and the given eigen decomposition of matrix Σ.

INTERPRETATION 1: reading the geometric result in revese shows that linear transformation "distort" Euclidean distances into Mahalanobis distances.

INTERPRETATION 2 (Euclidean embedding): a space with Mahalanobis metric can be isometrically embedded in a Euclidean space. This is a trivial spacial case of the Nash theorem on existence of Euclidean embeddings of more general (Riemannian) metric spaces.

Solution:

$$\begin{aligned} LHS &= \|X-\mu\|_{\Sigma}^2 \\ &= (X-\mu)^T\Sigma^{-1}(X-\mu) \end{aligned}$$

$$\begin{aligned} RHS &= \|Y-m\|^2 \\ &= (AX-A\mu)^T(AX-A\mu) \\ &= (x-\mu)^TA^TA(x-\mu) \end{aligned}$$

We can cancel out $(x-\mu)$ so we get,

$\Sigma^{-1} = A^TA$

$(U\Lambda U^T)^{-1} = A^TA$

$U^{-T}\Lambda^{-1}U^{-1} = A^TA$

Let $B = diag(\sqrt{\frac{1}{a_1}}, \sqrt{\frac{1}{a_2}}\dots\sqrt{\frac{1}{a_N}})$, where a_i is eigen values

Then,

$U^{-T}B^TBU^{-1} = A^TA$

$(BU^{-1})^TBU^{-1} = A^TA$

So $A = BU^{-1}$

Problem 3

Show algebraic equivalence between two non-parametric formulations for K-means (objectives $E(S)$ at the bottom of slide 67, Topic 9):

$$\sum_{k=1}^{K}\frac{\sum_{pq\in S^k}\|f_p-f_q\|^2}{2\ |S^k|} \quad = \quad const - \sum_{k=1}^{K}\frac{\sum_{pq\in S^k}\langle f_p, f_q\rangle}{|S^k|}$$

Solution:

Start with the numerator of the RHS,

$$\begin{aligned}\|f_p - f_q\|^2 &= (f_{p1} - f_{q1})^2 + (f_{p2} - f_{q2})^2 + \ldots + (f_{pN} - f_{qN})^2 \\ &= (f_{p1}^2 - 2f_{p1}f_{q1} + f_{q1}^2) + (f_{p2}^2 - 2f_{p2}f_{q2} + f_{q2}^2) + \ldots + (f_{pN}^2 - 2f_{pN}f_{qN} + f_{qN}^2) \\ &= (f_{p1}^2 + f_{q1}^2) + (f_{p2}^2 + f_{q2}^2) + \ldots + (f_{pN}^2 + f_{qN}^2) + (-2f_{p1}f_{q1}) + (-2f_{p2}f_{q2}) + \ldots + (-2f_{pN}f_{qN}) \\ &= (f_{p1}^2 + f_{q1}^2 + f_{p2}^2 + f_{q2}^2 + \ldots + f_{pN}^2 + f_{qN}^2) - 2(f_{p1}f_{q1} + f_{p2}f_{q2} + \ldots + f_{pN}f_{qN})\end{aligned}$$

Then if we substitute this into the original equation,

$$LHS = \sum_{k=1}^{K} \frac{\sum_{pq \in S^k} const - 2\langle f_p, f_q \rangle}{2\|S^k|}$$

take the const out, and it equals RHS

Problem 4 - OPTIONAL BONUS (a simple finite-dimensional version of Mercer theorem)

Let A be an $n \times n$ positive semi-definite matrix defining pairwise affinities between n ponts. Find a closed-form expression for n vectors ϕ_i (a so-called "Euclidean embedding") such that their Euclidean dot products agree with the given affinites, i.e. $\langle \phi_i, \phi_j \rangle = A_{ij}$ for all $1 \leq i, j \leq n$. You can assume known eigen-decomposition $A = Q\Lambda Q^T$ where $\Lambda = diag(\lambda_1, \ldots, \lambda_n)$ is a diagonal matrix of (non-negative!) eigen-values and Q is an orthogonal $n \times n$ matrix whose columns Q_i are unit eigen-vectors of A.

Solution:

$A = Q\Lambda Q^T$

$$A = \begin{bmatrix} q_{11} \ldots q_{1n} \\ \ldots \\ q_{n1} \ldots q_{nn} \end{bmatrix} \begin{bmatrix} \lambda_1 \ldots 0 \\ \ldots \\ 0 \ldots \lambda_n \end{bmatrix} \begin{bmatrix} q_{11} \ldots q_{n1} \\ \ldots \\ q_{1n} \ldots q_{nn} \end{bmatrix}$$

$$A = \begin{bmatrix} \lambda_1 q_{11} \ldots \lambda_n q_{1n} \\ \ldots \\ \lambda_1 q_{n1} \ldots \lambda_n q_{nn} \end{bmatrix} \begin{bmatrix} q_{11} \ldots q_{n1} \\ \ldots \\ q_{1n} \ldots q_{nn} \end{bmatrix}$$

$A_{ij} = \lambda_1 q_{i1} q_{j1} + \ldots + \lambda_n q_{in} q_{jn}$

$A_{ij} = \sqrt{\lambda_1} q_{i1} \sqrt{\lambda_1} q_{j1} + \ldots + \sqrt{\lambda_n} q_{in} \sqrt{\lambda_n} q_{jn}$

Therefore, $\phi_i = Q_i \sqrt{\lambda_i}$

Problem 5 - OPTIONAL BONUS (approximate low-dimenstional Euclidean embedding)

Assume that $\tilde{A}$ is a low-rank approximation of matrix A in problem 4 of given rank $m < n$. That is, $\tilde{A} = Q\Lambda_m Q^T$ where $\Lambda_m = diag(\lambda_1, \dots, \lambda_m, 0, \dots, 0)$ is a diagonal matrix of the largest m eigen values of A (a la Eckart–Young–Mirsky theorem, Topic 8). Using your solution for problem 2, specify a formula for "Euclidean embedding" $\{\tilde{\phi}_i\}$ such that $\langle \tilde{\phi}_i, \tilde{\phi}_j \rangle = \tilde{A}_{ij}$ and show that $\tilde{\phi}_i \in \mathcal{R}^m$.

Comment: basic K-means (Lloyd's algorithm) over such points $\{\tilde{\phi}_i\}$ can be used as an approximate algorithm for kernel clusterng (e.g. for average association criteria). This approach is an example of "spectral clustering", which uses eigen decomposition of the affinity matrix A.

Solution:

For A_{ij}, if $i > m, j > m$ the result becomes 0.

So $A_{ij} = \langle \tilde{\phi}_i, \tilde{\phi}_j \rangle$ where $\tilde{\phi}_i = \begin{bmatrix} \sqrt{\lambda_1} q_{i1} \\ \dots \\ \sqrt{\lambda_m} q_{im} \end{bmatrix}$

Problem 4 (K-means).

Subproblem 4.1

Implement K-means algorithm for clustering pixel features. Most of the work is already done for you, but you do get a chance to play with numpy and to evaluate empirical properties of K-means. Note that your implementation will be slow if you use double-loops to traverse the pixels. There will be deductions for such double-loops. You should learn how to use functions like $np.where$, $np.minimum$, $np.square$, $np.ogrid$ or others similar general functions that alow to avoid multi-loops over matrix (image) elements (pixels).

The provided code below only computes random pixel segments. You need to write code producing correct clusters and correct "means". To achive this you only need to complete implementation of functions $compute\_means$ and $compute\_labels$ inside "MyKmenasApp" corresponding to the two iterative steps in Lloyd's algorithm (as in "compute_k_means_clusters").

Your implementation of the main two steps of K-means algorithm should use RGBXY features. Relative contribution of "squared errors" from XY features must be set by parameter "weightXY" (or self.w inside MyKmeansApp), so that the squared error between RGBXY feture $F_p = [R_p, G_p, B_p, X_p, Y_p]$ at any pixel p and any given cluster mean $m = [R_m, G_m, B_m, X_m, Y_m]$ is

$$||F_p - m||^2 = (R_p - R_m)^2 + (G_p - G_m)^2 + (B_p - B_m)^2 + w \cdot (X_p - X_m)^2 + w \cdot (Y_p - Y_m)^2.$$

Fully implemented "KmeansPresenter" visulaizes the segmentation results (cluster labels mask) where each cluster is highlighted either by some random color (press r-key) or by the "mean" segment color (press m-key). All keys that "KmeansPresenter" responds to are as follows:

1. press 'i'-key for each (i)teration of K-means
2. press 'c'-key to run K-means to (c)onvergence (when energy improvement is less than given threshold)
3. press 'v'-key to run K-means to convergence with (v)isualization of each iteration
4. press 'r'-key to start over from (r)andom means
5. press 's'-key to change to a random (s)olid color-palette for displaying clusters
6. press 't'-key to change to a random (t)ransparent palette for displaying clusters
7. press 'm'-key to change to the (m)ean-color palette for displaying clusters

In [2]:
```
%matplotlib notebook
# loading standard modules
import numpy as np
import math
import matplotlib.pyplot as plt
from skimage import img_as_ubyte
from skimage.color import rgb2grey

# loading custom module (requires file asg1.py in the same directory as
 the notebook file)
from asg1_error_handling import Figure, KmeansPresenter
```

In [3]:
```
class MyKmeansApp:

    def __init__(self, img, num_clusters=2, weightXY=1.0):
        self.k = num_clusters
        self.w = weightXY
        self.iteration = 0   # iteration counter
        self.energy = np.infty  # energy - "sum of squared errors" (SSE)

        num_rows = self.num_rows = img.shape[0]
        num_cols = self.num_cols = img.shape[1]

        self.im = img

        self.means = np.zeros((self.k,5),'d') # creates a zero-valued (d
ouble) matrix of size Kx5
        self.init_means()

        self.no_label = num_clusters  # special label value indicating p
ixels not in any cluster (e.g. not yet)

        # mask "labels" where pixels of each "region" will have a unique
index-label (like 0,1,2,3,..,K-1)
        # the default mask value is "no-label" (K) implying pixels that
 do not belong to any region (yet)
        self.labels = np.full((num_rows, num_cols), fill_value=self.no_l
abel, dtype=np.int)

        self.fig = Figure()
        self.pres = KmeansPresenter(img, self)
        self.pres.connect_figure(self.fig)

    def run(self):
        self.fig.show()

    def init_means(self):
        self.iteration = 0           # resets iteration counter
        self.energy = np.infty       # and the energy

        poolX = range(self.num_cols)
        poolY = range(self.num_rows)

        # generate K random pixels (Kx2 array with X,Y coordinates in ea
ch row)
        random_pixels = np.array([np.random.choice(poolX,self.k),np.rand
om.choice(poolY,self.k)]).T

        for label in range(self.k):
            self.means[label,:3] = self.im[random_pixels[label,1],random
_pixels[label,0],:3]
            self.means[label,3] = random_pixels[label,0]
            self.means[label,4] = random_pixels[label,1]

    # This function compute average values for R, G, B, X, Y channel (fe
ature component) at pixels in each cluster
    # represented by labels in given mask "self.labels" storing indeces
 in range [0,K). The averages should be
```

```
    # saved in (Kx5) matrix "self.means". The return value should be the
number of non-empty clusters.
    def compute_means(self):
        labels = self.labels
        non_empty_clusters = 0

        # Your code below should compute average values for R,G,B,X,Y fe
atures in each segment
        # and save them in (Kx5) matrix "self.means". For empty clusters
set the corresponding mean values
        # to infinity (np.infty). Report the correct number of non-empty
clusters by the return value.
        clusters  = {}
        for i in range(self.k):
            temp = np.argwhere(labels==i)
            if temp.shape[0]:
                R = np.mean(self.im[temp[:,0], temp[:,1], 0])
                G = np.mean(self.im[temp[:,0], temp[:,1], 1])
                B = np.mean(self.im[temp[:,0], temp[:,1], 2])
                X = np.mean(temp[:,0])
                Y = np.mean(temp[:,1])
                non_empty_clusters += 1
            else:
                R = np.infty
                G = np.infty
                B = np.infty
                X = np.infty
                Y = np.infty
            self.means[i] = np.array([R, G, B, X, Y])

        return non_empty_clusters

    # The segmentation mask is used by KmeanPresenter to paint segments
 in distinct colors
    # NOTE: valid region labels are in [0,K), but the color map in Kmean
sPresenter
    #       accepts labels in range [0,K] where pixels with no_label=K a
re not painted/colored.
    def get_region_mask(self):
        return self.labels

    # This function computes optimal (cluster) index/label in range 0,
1,...,K-1 for pixel x,y based on
    # given current cluster means (self.means). The functions should sav
e these labels in "self.labels".
    # The return value should be the corresponding optimal SSE.
    def compute_labels(self):
        shape = (self.num_rows,self.num_cols)
        opt_labels = np.full(shape, fill_value=self.no_label, dtype=np.i
nt) # HINT: you can use this array to store and update

# currently the best label for each pixel.

        min_dist = np.full(shape, fill_value=np.inf)  # HINT: you can us
e this array to store and update
                                                      # the (squared) di
stance from each pixel to its current "opt_label".
```

```
                                                    # use 'self.w' as
 a relative weight of sq. errors for X and Y components

        # Replace the code below by your code that computes "opt_labels"
array of labels in range [0,K) where
        # each pixel's label is an index 'i' such that self.mean[i] is t
he closest to R,G,B,X,Y values of this pixel.
        # Your code should also update min_dist so that it contains the
 optmail squared errors
        opt_labels = np.random.choice(range(self.k),shape)
        for i in range(opt_labels.shape[0]):
            for j in range(opt_labels.shape[1]):
                for k in range(self.k):
                    sqr_d = ((self.im[i][j][0]-self.means[k][0])**2)+((s
elf.im[i][j][1]-self.means[k][1])**2)+((self.im[i][j][2]-self.means[k][2
])**2)+((i-self.means[k][3])**2)+((j-self.means[k][4])**2)
                    if (sqr_d < min_dist[i][j]):
                        min_dist[i][j] = sqr_d
                        opt_labels[i][j] = k

        # update the labels based on opt_labels computed above
        self.labels = opt_labels

        # returns the optimal SSE (corresponding to optimal clusters/lab
els for given means)
        return min_dist.sum()

    # The function below is called by "on_key_down" in KmeansPresenter".
    # It's goal is to run an iteration of K-means procedure
    # updating the means and the (segment) labels
    def compute_k_means_clusters(self):
        self.iteration += 1

        # the main two steps of K-means algorithm
        energy = self.compute_labels()
        num_clusters = self.compute_means()

        # computing improvement and printing some information
        num_pixels = self.num_rows*self.num_cols
        improve_per_pixel = (self.energy - energy)/num_pixels
        energy_per_pixel = energy/num_pixels
        self.energy = energy

        self.fig.ax.text(0, -8,   # text location
                         'iteration = {:_>2d},  clusters = {:_>2d},  SS
E/p = {:_>7.1f},   improve/p = {:_>7.3f}    '.format(
                         self.iteration,       num_clusters,        ene
rgy_per_pixel,    improve_per_pixel),
                         bbox={'facecolor':'white', 'edgecolor':'none'
})

        return improve_per_pixel
```

Subproblem 4.2

Use K-means to generate 3-4 representative results (you can use your own images) with color quantization and superpixels. Experiment with different values of parameter K (in the range 2-80). Compare representative values of optimal SSE for smaller and larger K and explain the observed differences. Add more cells (code and/or text) as necessary.

In [3]:
```
img = plt.imread('images/rose.bmp')
app = MyKmeansApp(img, num_clusters=2, weightXY=2.0)
app.run()
```

K-means

iteration = 11, clusters = _2, SSE/p = 19145.6, improve/p = __0.001

In [4]:
```
img = plt.imread('images/rose.bmp')
app = MyKmeansApp(img, num_clusters=10, weightXY=2.0)
app.run()
```

K-means

iteration = 51, clusters = _6, SSE/p = _8164.7, improve/p = __0.001

In [4]:
```
img = plt.imread('images/rose.bmp')
app = MyKmeansApp(img, num_clusters=20, weightXY=2.0)
app.run()
```

K-means

iteration = _3, clusters = 12, SSE/p = _5478.1, improve/p = 1671.779

In [6]:
```
img = plt.imread('images/rose.bmp')
app = MyKmeansApp(img, num_clusters=80, weightXY=2.0)
app.run()
```

K-means

iteration = _3, clusters = 40, SSE/p = _2469.8, improve/p = 1036.904

As K becomes largers, we reach a smaller SSE more quickly. We can see in the first 2 figures where K is small, it took less iterations to converge, but the resulting SSE is huge. In the last 2 figures where K is large, it took too long to converge so it is only run for 3 iterations, and we already can see the SSE being smaller than the first 2 converged graph. This is because for larger K, there are more labels to pick from and it is easier to get a smaller mean (closer to a particular pixel).

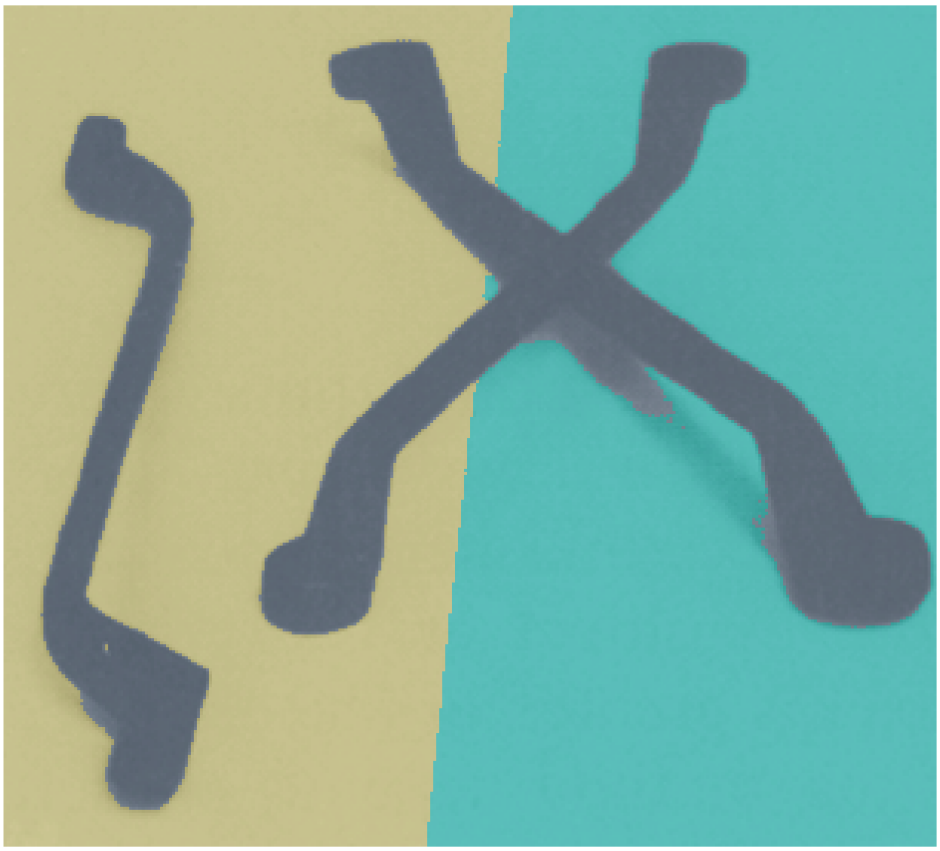
Subprobelm 4.3

Demonstrate sensitivity of K-means to local minima (you can use your own images). Show 2-3 different solutions for different random initial means and display the corresponding values of the K-means energy. Add more cells (code and/or text) as necessary. Play wth different weights w and different number of clusters, different images.

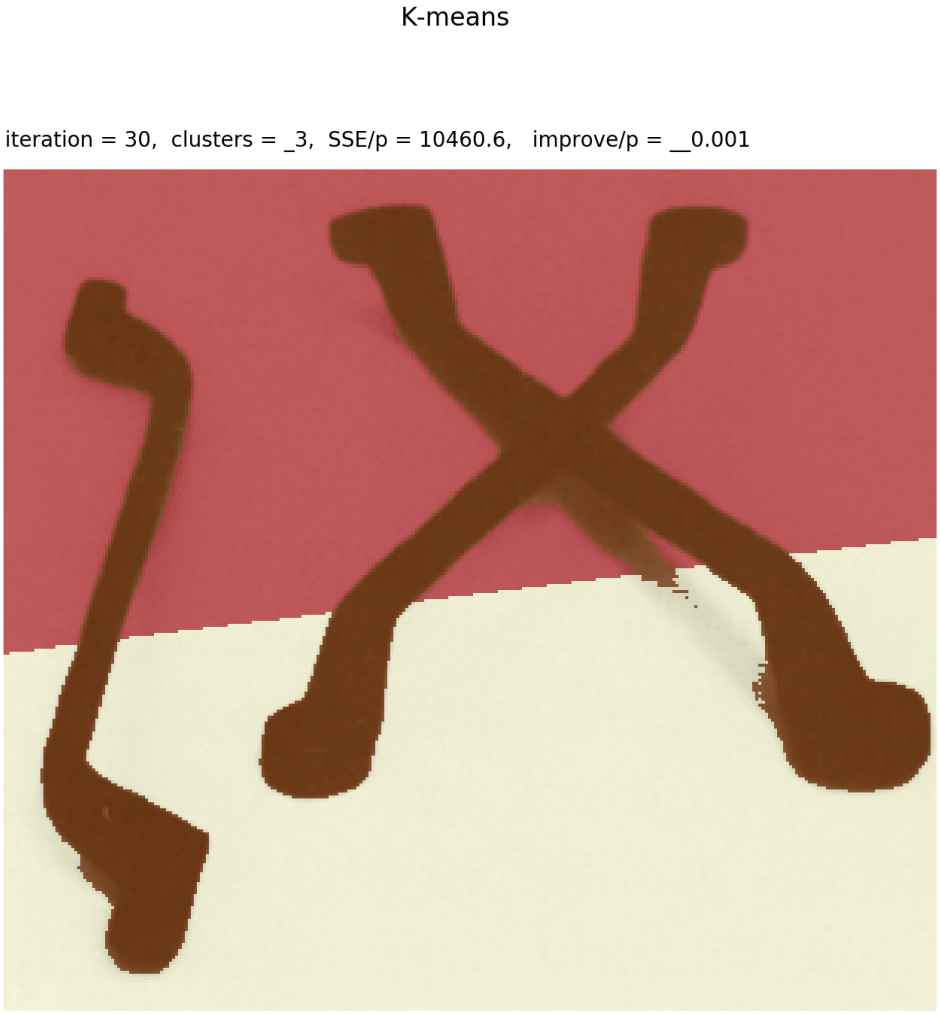
In [19]:
```
img = plt.imread('images/tools.bmp')
app = MyKmeansApp(img, num_clusters=3, weightXY=0.0)
app.run()
```

K-means

iteration = 25, clusters = _3, SSE/p = _9882.9, improve/p = __0.001



In [21]:
```
img = plt.imread('images/tools.bmp')
app = MyKmeansApp(img, num_clusters=3, weightXY=0.0)
app.run()
```



The 2 K-means pictures are exactly the same with the same setup. We can see the resulting graphs are clustered differently. One with vertical dividing line and one with horizontal dividing line.

In [ ]:
```

```