Kernel K-means

Kernel k-means is an approach to k-means algorithm, but mapping the data into higher degree dimentions.
And the mapping-function called `Kernel`.

K-means algorithm is that after comparing the data similarity, we cluster the more similarity datas to the same group.
`K` means that there are k number of group to cluster.

For the regular K-means, we use the following formula to compare and cluster data.

$arg\ min_{(C_1, \mu_1)...(C_k, \mu_k)} \sum_{i=1}^{k} \sum_{x_j \in C_i} ||x_j - \mu_i||^2$

How about kernel k-means?

we transfrom the data to the higher degree, and do the same k-means algorithm on it.

$arg\ min_{(C_1, \mu_1^\phi)...(C_k, \mu_k^\phi)} \sum_{i=1}^{k} \sum_{\phi(x_j) \in C_i} ||\phi(x_j) - \mu_i^\phi||^2$

$$||\phi(x_j) - \mu_i^\phi||^2 = \phi^T(x_j)\phi(x_j) - 2\phi^T(x_j)\frac{1}{|C_k|}\sum_{x_n \in c_k}\phi(x_n) + \frac{1}{|C_k|^2}\sum_{x_p \in C_k}\sum_{x_q \in C_k}\phi^T(x_p)\phi(x_q)$$

$$= K(x_j, x_j) - \frac{2}{|C_k|}\sum_{x_n \in c_k}K(x_j, x_n) + \frac{1}{|C_k|^2}\sum_{x_p \in C_k}\sum_{x_q \in C_k}K(x_p, x_q)$$

Instead of update μ_k in the k-means algorithm, update only C_k in the kernel k-means algorithm.

The Work

- kernel function: $e^{-\gamma_1 ||S(x) - S(x')||^2} \times e^{-\gamma_2 ||C(x) - C(x')||^2}$
- Input data: Two 100*100 images

Step 1

Prepare image data for precompute gram matrix (`kernel`)

```python
def img_formater(img):
    n = img.shape[0]*img.shape[1]
    spatial_data = []
    color_data = []
    for i in range(img.shape[0]):
        for j in range(img.shape[1]):
            spatial_data.append([i, j])
            color_data.append(img[i][j])
    return np.array(spatial_data), np.array(color_data, dtype=int)


img = imageio.imread(img_path)
spatial_data, color_data = img_formater(img)
```

Step 2

Compute gram matrix

> At the first, I implement this part by for-loop which is a trivial way.
> But I found that it is too time-consuming! Because there are 10^4 data points for our input data, and we need to calculate a $10^4 \times 10^4$ gram matrix.
> So I replace my for-loop implementation into matrix-computation for efficient.

$$euclidean^2 = ||u - v||^2 = (u - v)^T(u - v) = ||u||^2 - 2u^Tv + ||v||^2$$

Above formula is suitable for vector which is stroed in $d \times 1$ matrix.
But in this work, the vectors are stored in $1 \times d$ matrix, so I applied the following formula revised.

$$euclidean^2 = ||u - v||^2 = (u - v)(u - v)^T = ||u||^2 - 2uv^T + ||v||^2$$

And for the performance, I take all vectors into one matrix D which is $n \times d$ for calculate gram matrix G which is $n \times n$, let E as the euclidean matrix

$$D = \begin{bmatrix} V_1 \\ V_2 \\ \ldots \\ V_{n-1} \\ V_n \end{bmatrix}_{n \times d} , V_i = \begin{bmatrix} v_{i1} & v_{i2} & \ldots & v_{id} \end{bmatrix}_{1 \times d}$$

$$E = \begin{bmatrix} ||V_1, V_1||^2 & ||V_1, V_2||^2 & \ldots & ||V_1, V_n||^2 \\ ||V_2, V_1||^2 & ||V_2, V_2||^2 & \ldots & ||V_2, V_n||^2 \\ \vdots & \vdots & \ldots & \vdots \\ ||V_n, V_1||^2 & ||V_n, V_2||^2 & \ldots & ||V_n, V_n||^2 \end{bmatrix}$$

$$= \begin{bmatrix} ||V_1||^2 - 2V_1V_1^T + ||V_1||^2 & \ldots & ||V_1||^2 - 2V_1V_n^T + ||V_n||^2 \\ \vdots & \ldots & \vdots \\ ||V_n||^2 - 2V_nV_1^T + ||V_1||^2 & \ldots & ||V_n||^2 - 2V_nV_n^T + ||V_n||^2 \end{bmatrix}$$

$$= \begin{bmatrix} ||V_1||^2 & \ldots & ||V_1||^2 \\ ||V_2||^2 & \ldots & ||V_2||^2 \\ \vdots & \ldots & \vdots \\ ||V_n||^2 & \ldots & ||V_n||^2 \end{bmatrix} - 2DD^T + \begin{bmatrix} ||V_1||^2 & \ldots & ||V_1||^2 \\ ||V_2||^2 & \ldots & ||V_2||^2 \\ \vdots & \ldots & \vdots \\ ||V_n||^2 & \ldots & ||V_n||^2 \end{bmatrix}^T$$

$$= D^2 \begin{bmatrix} 1 & 1 & \ldots & 1 \\ \vdots & \vdots & \ldots & \vdots \\ 1 & 1 & \ldots & 1 \end{bmatrix}_{d \times n} - 2DD^T + \begin{bmatrix} 1 & 1 & \ldots & 1 \\ \vdots & \vdots & \ldots & \vdots \\ 1 & 1 & \ldots & 1 \end{bmatrix}_{n \times d} (D^T)^2$$

So we could calculate G as $G = e^E$ without for-loop.

```python
#Origianl trvial implementation
def rbf_img(u, v, g=0.0001):
    s_dis = scipy.spatial.distance.euclidean(u[0], v[0])
    c_dis = scipy.spatial.distance.euclidean(u[1], v[1])
    return math.exp(-1*g*s_dis**2 - g*c_dis**2)

def gram_matrix(data, path, kernel=rbf_img):
    gram = np.ones((len(data), len(data)))
    for i in range(len(data)):
        for j in range(i, len(data)):
            gram[i][j] = kernel(data[i], data[j])
            gram[j][i] = gram[i][j]
    return gram


#-----------------------------------------------------------------#

#New implementation using matrix computation
def euclidean(self, u, v):
    #This method is defined in kmeans.euclidean
    return np.matmul(u**2, np.ones((u.shape[1],v.shape[0]))) \
        -2*np.matmul(u, v.T) \
```

```
        +np.matmul(np.ones((u.shape[0], v.shape[1])), (v.T)**2)

def rbf(u, v, g=10**-4):
    return np.exp(-1*g*kmeans.euclidean(kmeans, u, v))


gram = rbf(spatial_data, spatial_data) \
    * rbf(color_data, color_data)
```

Step 3

Run Kernel K-means, recall the formula:

$$Let\ s_{jk} = ||\phi(x_j) - \mu_k^\phi||^2 = K(x_j, x_j) - \frac{2}{|C_k|} \sum_{x_n \in c_k} K(x_j, x_n) + \frac{1}{|C_k|^2} \sum_{x_p \in C_k} \sum_{x_q \in C_k} K(x_p, x_q)$$

we compare for the $||\phi(x_j) - \mu_k^\phi||^2$ for measuring the distance between the kth cluster and the jth mapped data point at every vector x_j on cluster C_k, and for the every C_k, the first terms are the same, so we could ignore it directly. I still use matrix computaion on this part, and I would explain the procedure of my derivation as below.

For each x_j, we have k values of corresponding to the kth cluster (denoted by s_{jk}), and we would go throgh all datas, which means that we have $n \times k$ values in totally and it is suitable for matrix computation!
Let $S_{n \times k}$ is the distance matrix as mentioned (which is `dis` in the code segment).

$$s_{jk} = \frac{2}{|C_k|} \sum_{x_n \in c_k} K(x_j, x_n) + \frac{1}{|C_k|^2} \sum_{x_p \in C_k} \sum_{x_q \in C_k} K(x_p, x_q)$$

$$= \frac{2}{|C_k|} \begin{bmatrix} K(x_j, x_1) & \cdots & K(x_j, x_n) \end{bmatrix} C_k + \frac{1}{|C_k|^2} C_k{}^T G C_k$$

s_{jk} is the jth row k^{th} col element of $S_{n \times k}$, means the distance between the jth mapped data point and the kth cluster.

I want to do a matrix computation instead of n times computation at each data point. So I need expand the above euation to from 1×1 to $n \times k$.

$$S_{n \times k} = \frac{2}{|C|} \begin{bmatrix} K(x_1, x_1) & K(x_1, x_2) & \cdots & K(x_1, x_n) \\ K(x_2, x_1) & K(x_2, x_2) & \cdots & K(x_2, x_n) \\ \vdots & \vdots & \cdots & \vdots \\ K(x_n, x_1) & K(x_n, x_2) & \cdots & K(x_n, x_n) \end{bmatrix}_{n \times n} C_{n \times k}$$

$$+ \frac{1}{|C|^2} \begin{bmatrix} 1 & 1 & \cdots & 1 \\ \vdots & \vdots & \cdots & \vdots \\ 1 & 1 & \cdots & 1 \end{bmatrix}_{n \times k} C_{n \times k}{}^T G_{n \times n} C_{n \times k}$$

$$S_{n \times k} = \frac{2}{|C|} G_{n \times n} C_{n \times k} + \frac{1}{|C|^2} \begin{bmatrix} 1 & \cdots & 1 \\ \vdots & \cdots & \vdots \\ 1 & \cdots & 1 \end{bmatrix}_{n \times k} C_{n \times k}{}^T G_{n \times n} C_{n \times k}$$

For the second term, we only need the diagonal elements, so in the `python` code, I multiply use a diagonal identity matrix `np.eye()`.

And there are many varies of initialization method, e.g. `k-means++` which decide the initial cluster by maxmizing their distance. I would compare `k-means++` and the traditional way at Result part.

```python
def __get_init(self, method):
        if method == 'kmeans++':
            return self.__kmeanspp
        elif method == 'default':
            return self.__traditional
        else:
            print('ERROR: \'{}\' is not a pre-defined initialize
method'.format(method))
            return exit(0)

def __kmeanspp(self):
    n, d = self.data.shape
    centers = np.array([self.data[np.random.randint(n), :d]])
    for i in range(self.k-1):
        dist = self.euclidean(self.data, centers)
        dist = np.min(dist, axis=1)
        next_center = np.argmax(dist, axis=0)
        centers = np.vstack((centers, self.data[next_center, :]))
    return centers

def __traditional(self):
    return np.array(self.data[np.random.choice(self.data.shape[0],
size=self.k, replace=False), :])

def __kernel_trick(self, gram, ck):
    c_count = np.sum(ck, axis=0)
    dist = -2*np.matmul(gram, ck)/c_count + \
        np.matmul(np.ones(ck.shape), (np.matmul(ck.T, np.matmul(gram,
ck)))*np.eye(ck.shape[1]))/(c_count**2)
    return dist

def run(self):
    #initial cluster
    centers = self.init()
    dist = self.euclidean(self.data, centers)
    ck = np.zeros((self.data.shape[0], self.k))
    ck[np.arange(dist.shape[0]), np.argmin(dist, axis=1)] = 1

    record = []
    record.append(ck)
    record_iter = 0
    for i in range(self.max_iter):
```

```python
            #E-step
            if self.is_kernel:
                dist = self.__kernel_trick(self.data, ck)
            else:
                dist = self.euclidean(self.data, centers)

            #M-step
            update_ck = np.zeros(dist.shape)
            update_ck[np.arange(dist.shape[0]),np.argmin(dist, axis=1)] = 1
            delta_ck = np.count_nonzero(np.abs(update_ck - ck))
            update_centers = np.matmul(update_ck.T, self.data)/np.sum(update_ck,
axis=0, keepdims=True).T

            record.append(update_ck)
            if delta_ck == 0:
                self.converge -= 1
                if self.converge == 0:
                    record_iter = i+1
                    if self.keep_log == False:
                        break

            ck = update_ck
            centers = update_centers
        return record, record_iter
```

Step 4

Visualization

```python
k_visual = colors.to_rgba_array(['tab:blue', \
                        'tab:orange', \
                        'tab:green', \
                        'tab:red', \
                        'tab:purple', \
                        'tab:brown'])
def visualizer(record, save_path, figsize=(100,100,4)):
    gif = []
    for i in range(len(record)):
        c_id = np.argmax(record[i], axis=1)
        img = np.zeros(figsize, dtype=np.uint8)
        for j in range(c_id.shape[0]):
            m, n = (int(j/100), int(j%100))
            img[m][n] = 255*k_visual[c_id[j]]
        gif.append(img)
    imageio.mimsave(save_path, gif)

def merge_gifs(gifs, max_fram, id, method):
    gif = []
    for i in range(len(gifs)):
```

```
        gif.append(imageio.get_reader('output/'+gifs[i]))


    new_gif = imageio.get_writer('output/image'+str(id)+'_'+method+'.gif')


    if max_fram + 5 < 100:
        max_fram += 5
    for frame_number in range(max_fram):
        img = []
        for i in range(len(gif)):
            img.append(gif[i].get_next_data())
        new_image = np.hstack(img)
        new_gif.append_data(new_image)
    for i in range(len(gif)):
        gif[i].close()
    new_gif.close()
```

Screen shot

```
processing image1...
running kernel k-means (k = 2, default)..........[complete at [14] iterations]
visualizing..........[complete]
running kernel k-means (k = 2, kmeans++)..........[complete at [10] iterations]
visualizing..........[complete]
faster.....................[kmeans++]
running kernel k-means (k = 3, default)..........[complete at [16] iterations]
visualizing..........[complete]
running kernel k-means (k = 3, kmeans++)..........[complete at [11] iterations]
visualizing..........[complete]
faster.....................[kmeans++]
running kernel k-means (k = 4, default)..........[complete at [41] iterations]
visualizing..........[complete]
running kernel k-means (k = 4, kmeans++)..........[complete at [37] iterations]
visualizing..........[complete]
faster.....................[kmeans++]
running kernel k-means (k = 5, default)..........[complete at [45] iterations]
visualizing..........[complete]
running kernel k-means (k = 5, kmeans++)..........[complete at [62] iterations]
visualizing..........[complete]
faster.....................[tradition]
running kernel k-means (k = 6, default)..........[complete at [67] iterations]
visualizing..........[complete]
running kernel k-means (k = 6, kmeans++)..........[complete at [37] iterations]
visualizing..........[complete]
faster.....................[kmeans++]
processing image2...
running kernel k-means (k = 2, default)..........[complete at [17] iterations]
visualizing..........[complete]
running kernel k-means (k = 2, kmeans++)..........[complete at [19] iterations]
visualizing..........[complete]
faster.....................[tradition]
running kernel k-means (k = 3, default)..........[complete at [42] iterations]
visualizing..........[complete]
running kernel k-means (k = 3, kmeans++)..........[complete at [23] iterations]
visualizing..........[complete]
faster.....................[kmeans++]
running kernel k-means (k = 4, default)..........[complete at [56] iterations]
visualizing..........[complete]
running kernel k-means (k = 4, kmeans++)..........[complete at [23] iterations]
visualizing..........[complete]
faster.....................[kmeans++]
running kernel k-means (k = 5, default)..........[complete at [48] iterations]
visualizing..........[complete]
running kernel k-means (k = 5, kmeans++)..........[complete at [27] iterations]
visualizing..........[complete]
faster.....................[kmeans++]
running kernel k-means (k = 6, default)..........[complete at [33] iterations]
visualizing..........[complete]
running kernel k-means (k = 6, kmeans++)..........[complete at [38] iterations]
visualizing..........[complete]
faster.....................[tradition]
```

Result

- image1 (k=2, 3, 4, 5, 6)
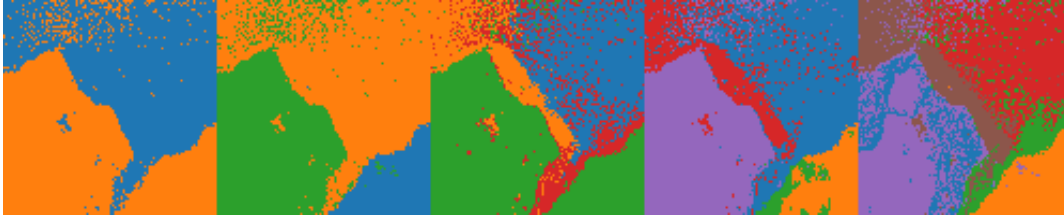
Original image:

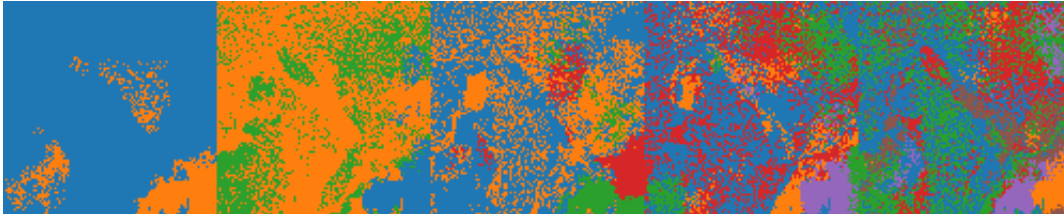$\gamma = 10^{-6}$:

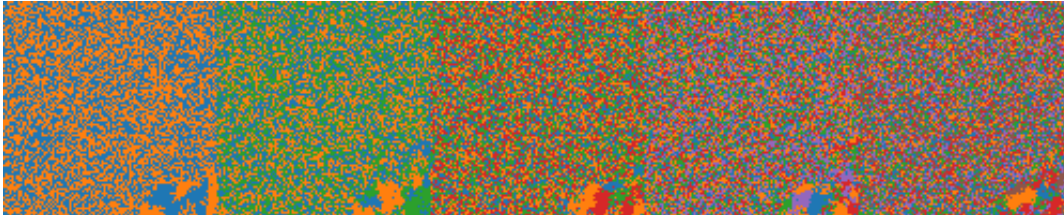$\gamma = 10^{-5}$:

$\gamma = 10^{-4}$:
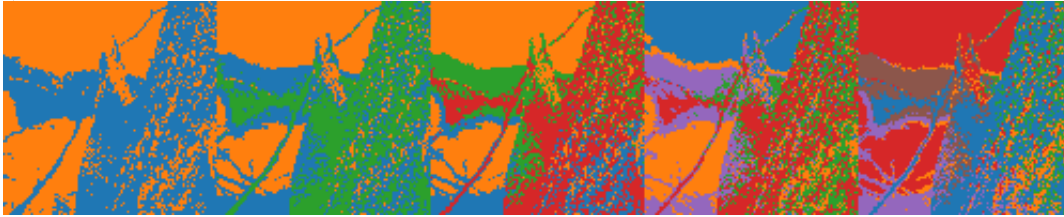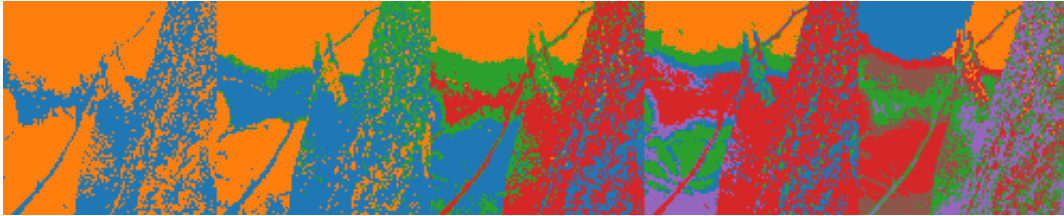

$\gamma = 10^{-3}$:

$\gamma = 10^{-2}$:

$\gamma = 10^{-1}$:

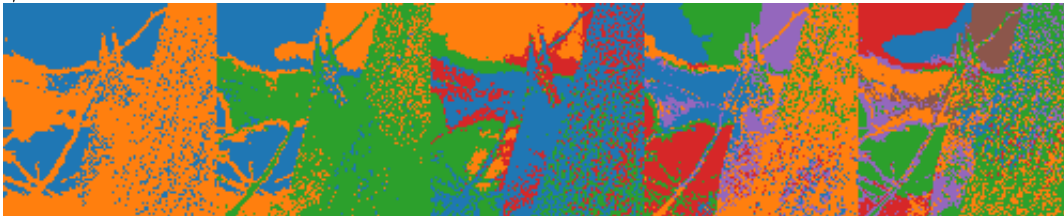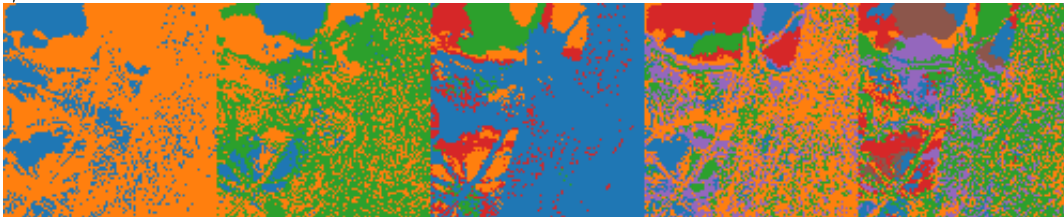- image2 (k=2, 3, 4, 5, 6)

$\gamma = 10^{-6}$:

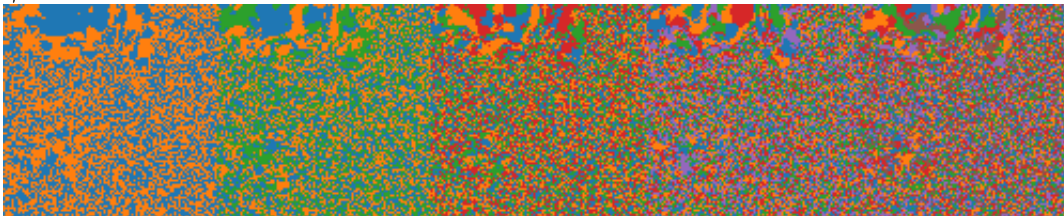$\gamma = 10^{-5}$:

$\gamma = 10^{-4}$:

$\gamma = 10^{-3}$:

$\gamma = 10^{-2}$:

$\gamma = 10^{-1}$:

For the value of γ parameter in the RBF kernel, I found that when γ is lower, the effect of clustering is larger.

The reason I thought is that the lower γ hints that the higer σ of the Gaussion distribution (i.e., In this work, this distribution is the distance between the point and the center of the cluster.), but too lower γ (i.e., higher σ) may cause underfitting.

- Traditional v.s. K-means++
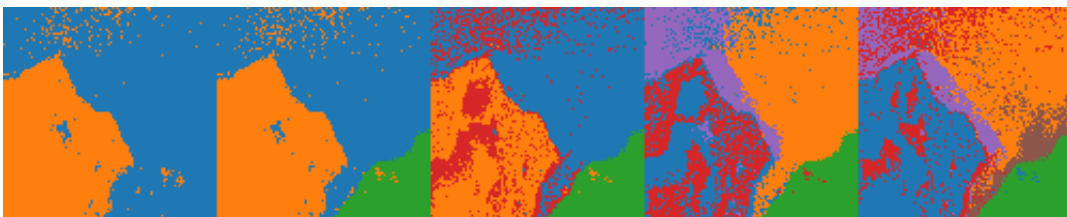 - Both two image are $\gamma = 10^{-4}$:
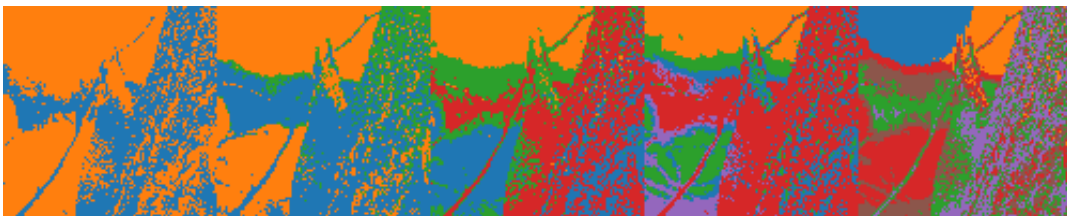

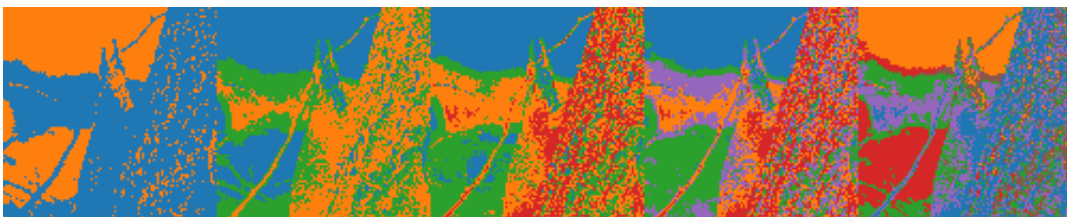   Traditional:
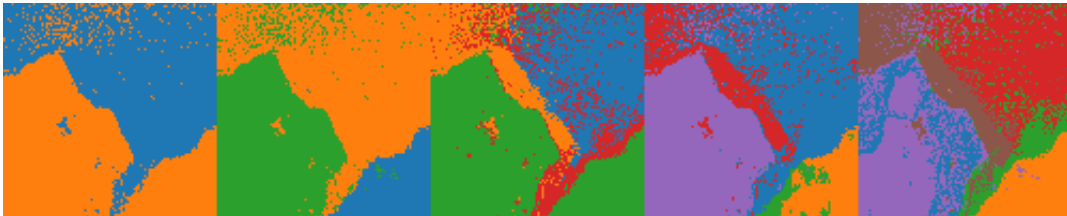
    K-means++:

    Traditional:

    K-means++:

At first, I thought that the K-mean++ just more faster than the tradition according the number of iteration (shown at [Screen shot part](#)). But when the result of image2 produced, we could see that the sky area and the rabbit could be clustered in the different when `k=6` ! So I believe that Kmeans++ is more powerful in this work.