Учреждение образования

«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ»

Кафедра информатики

Отчет по предмету:

«Машинное обучение»

По лабораторной работе №6

«Кластеризация»

Выполнил: Сенькович Дмитрий Сергеевич

магистрант кафедры информатики

группы №858642

Проверил: Стержанов Максим Валерьевич

доцент, кандидат технических наук

Минск 2019

**Оглавление**

[1 Введение 2](#_Toc25781306)

[2 Кластеризация точек 4](#_Toc25781307)

[2.1 Постановка задачи 4](#_Toc25781308)

[2.2 Чтение данных 4](#_Toc25781309)

[2.3 График исходных данных 4](#_Toc25781310)

[2.4 Реализация функции случайной инициализации центроидов 4](#_Toc25781311)

[2.5 Реализация функции принадлежности к кластерам 5](#_Toc25781312)

[2.6 Реализация функции пересчета центров кластеров 5](#_Toc25781313)

[2.7 Реализация алгоритма K-means 5](#_Toc25781314)

[2.8 Поиск лучшего решения и графики 7](#_Toc25781315)

[3 Сжатие изображения 13](#_Toc25781316)

[3.1 Постановка задачи 13](#_Toc25781317)

[3.2 Чтение данных 13](#_Toc25781318)

[3.3 Исходное изображение 13](#_Toc25781319)

[3.4 Построение сжатия 14](#_Toc25781320)

[3.5 Оценка качества сжатия 14](#_Toc25781321)

[3.6 Сжатие на другом изображении 15](#_Toc25781322)

[3.7 Иерархическая кластеризация 17](#_Toc25781323)

# 1 Введение

В предыдущих лабораторных работах рассматривались только методы обучения с учителем. В данном лабораторной работе затрагивается еще одна большая область машинного обучения - обучение без учителя. Обучение без учителя представляет собой обучение на данных без меток, то есть правильных ответов.

Одна из таких задач - задача кластеризации, то есть разделения данных по группам. Один из простейших алгоритмов решения это задачи - алгоритм К-средних. Несмотря на его простоту, алгоритм широко распространен на практике.

Алгоритм состоит из трех шагов:

* Выбор точек в качестве начальных центроидов: выбираются k точек (количество желаемых кластеров). Лучше всего выбирать точки случайным образом.
* Разделение точек по кластером относительно близости точек к каждому из центроидов, используя некоторую меру метрики, например, евклидову метрику, то есть расстояние между точками.
* Изменение центроидов. Каждый центроид усредняется в соответствии с точками, попавшими в его кластер.

Второй и третий шаг повторяются, пока координаты центроидов перестанут меняться или некоторое количество итераций алгоритма не будет достигнуто.

Алгоритм находит локальный оптимум, поэтому следует использовать алгоритм несколько раз на нескольких начальных центроидах для получения лучшего разбиения.

Лучшее разбиение определяется функцией стоимости, которая представляет из себя усредненную сумму дальностей точек в кластере от центроида этого кластера:

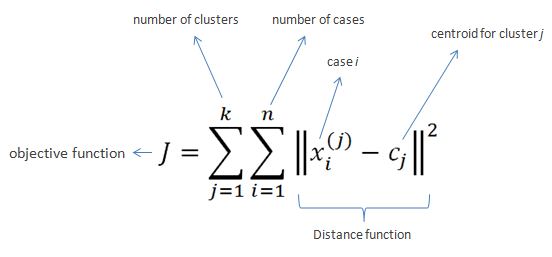


Рисунок 1.1. Функция стоимости в K-means алгоритме

# 2 Кластеризация точек

### 2.1 Постановка задачи

Имеется выборка точек, которые нужно кластеризовать.

### 2.2 Чтение данных

Считаем данные:

data = loadmat("ex6data1.mat")

x = data['X']

### 2.3 График исходных данных

Построим данные:

plt.scatter(x[:, 0], x[:, 1])

plt.show()

Полученный график:

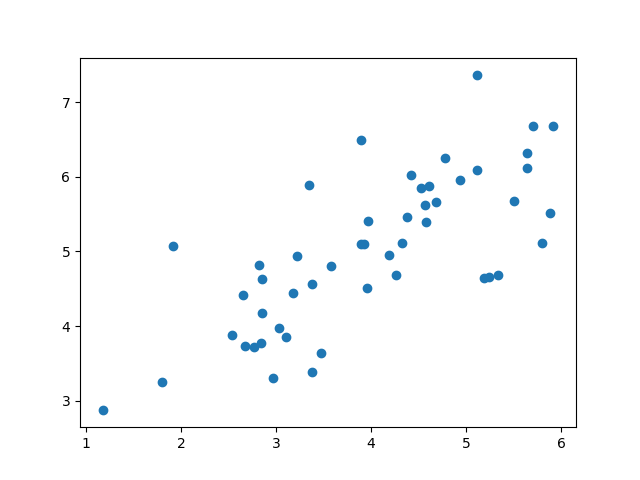


Рисунок 2.3.1. Исходные данные

### 2.4 Реализация функции случайной инициализации центроидов

Реализуем функцию:

def assign\_random\_centroids(x, K):

m = x.shape[0]

centroid\_indices = random.sample(range(1, m), K)

return np.array([x[centroid\_index] for centroid\_index in centroid\_indices])

### 2.5 Реализация функции принадлежности к кластерам

Логика определения кластера реализована следующим образом (выбирается центроид с наименьшим расстоянием до него, используя евклидову метрику):

distance = np.power(last\_centroids - xi.reshape(1, -1), 2).sum(axis=1)

closest\_centroid\_index = np.argmin(distance)

centroid\_indices\_per\_sample[i] = closest\_centroid\_index

### 2.6 Реализация функции пересчета центров кластеров

Приведем листинг кода, выполняющий данную логику:

def pick\_centroid\_neighbours(x, centroid\_indices\_per\_sample, centroid\_index):

centroid\_neighbours\_indices = [i for i in range(x.shape[0]) if centroid\_indices\_per\_sample[i] == centroid\_index]

return np.array([x[i] for i in centroid\_neighbours\_indices])

centroids = np.zeros(shape = (K, x.shape[1]))

for k in range(K):

neighbours = pick\_centroid\_neighbours(x, centroid\_indices\_per\_sample, k)

centroids[k] = neighbours.sum(axis=0) / (1 if len(neighbours) == 0 else len(neighbours))

centroid\_history[k].append(centroids[k])

Для каждого центроида выбираются точки, принадлежащие кластеру, и в качестве новых координат центроида кластера берется среднее значение точек кластера по каждой из координат.

### 2.7 Реализация алгоритма K-means

Для реализации алгоритма осталось реализовать функцию стоимости:

def cost(x, centroids, centroid\_indices\_per\_sample):

K = centroids.shape[0]

m = x.shape[0]

cost = 0

for k in range(K):

centroid = centroids[k]

neighbours = pick\_centroid\_neighbours(x, centroid\_indices\_per\_sample, k)

if not len(neighbours):

continue

distance = np.power(neighbours - centroid.reshape(1, -1), 2).sum(axis=1)

cost += distance.sum(axis=0)

return cost / m

Реализуем сам алгоритм. Он будет также сохранять траекторию движения центроидов для построения графиков:

def k\_means(x, K):

last\_centroids = assign\_random\_centroids(x, K)

m = x.shape[0]

centroid\_indices\_per\_sample = [0] \* m

centroid\_history = [[last\_centroids[0]], [last\_centroids[1]], [last\_centroids[2]]]

for \_ in range(k\_means\_iterations\_count):

for i in range(m):

xi = x[i]

distance = np.power(last\_centroids - xi.reshape(1, -1), 2).sum(axis=1)

closest\_centroid\_index = np.argmin(distance)

centroid\_indices\_per\_sample[i] = closest\_centroid\_index

centroids = np.zeros(shape = (K, x.shape[1]))

for k in range(K):

neighbours = pick\_centroid\_neighbours(x, centroid\_indices\_per\_sample, k)

centroids[k] = neighbours.sum(axis=0) / (1 if len(neighbours) == 0 else len(neighbours))

centroid\_history[k].append(centroids[k])

last\_centroids = centroids

for k in range(K):

centroid\_history[k] = np.array(centroid\_history[k])

return last\_centroids, centroid\_history, cost(x, last\_centroids, centroid\_indices\_per\_sample), centroid\_indices\_per\_sample

И для поиска лучшего ответа реализуем метод, который будет искать лучшее разбиение в соответствии со значением функции стоимости:

def find\_best\_clusterization(x, K):

best\_cost = 100000000000000000000

best\_centroids = []

best\_centroid\_history = []

best\_centroid\_indices\_per\_sample = []

for \_ in range(algorithm\_iterations\_count):

centroids, centroid\_history, cost, centroid\_indices\_per\_sample = k\_means(x, K)

if cost < best\_cost:

best\_cost = cost

best\_centroids = centroids

best\_centroid\_history = centroid\_history

best\_centroid\_indices\_per\_sample = centroid\_indices\_per\_sample

return best\_centroids, best\_centroid\_history, best\_cost, best\_centroid\_indices\_per\_sample

### 2.8 Поиск лучшего решения и графики

Поиск лучшего решения будем производить, разбивая точки на три кластера, запуская алгоритм 100 раз с 5 итерациями алгоритма (для данной выборки не нужно много итераций, да и на графиках будет видно, что с большим количеством итераций центроиды уже не двигаются).

Перед поиском лучшего решения подготовимся и реализуем 4 функции: построения графика данных с центроидами, построения графика с центроидами в цвете, построения графика траекторий всех центроидов и построения графика траектории центроида в ходе алгоритма:

def plot\_data\_with\_centroids(x, centroids):

plt.scatter(x[:, 0], x[:, 1])

plt.scatter(centroids[:, 0], centroids[:, 1], c='g')

plt.show()

def plot\_clusters\_colorized(x, centroid\_indices\_per\_sample, centroids):

for k in range(3):

color = centroid\_colors[k]

neighbours = pick\_centroid\_neighbours(x, centroid\_indices\_per\_sample, k)

plt.scatter(neighbours[:, 0], neighbours[:, 1], c=color)

centroid = centroids[k]

plt.scatter(centroid[0], centroid[1], c=color, s=150)

plt.show()

def plot\_centroid\_trajectory(centroid\_history, k):

labels = [\*range(1, len(centroid\_history[0] + 1))]

plt.plot(centroid\_history[k][:, 0], centroid\_history[k][:, 1], c=centroid\_colors[k])

for i in range(len(labels)):

label = str(labels[i])

plt.text(centroid\_history[k][i, 0], centroid\_history[k][i, 1], label, fontsize=9)

plt.show()

def plot\_centroids\_trajectory(centroid\_history):

labels = [\*range(1, len(centroid\_history[0] + 1))]

for k in range(3):

plt.plot(centroid\_history[k][:, 0], centroid\_history[k][:, 1], c=centroid\_colors[k])

for i in range(len(labels)):

label = str(labels[i])

plt.text(centroid\_history[k][i, 0], centroid\_history[k][i, 1], label, fontsize=9)

plt.show()

Приступим к поиску лучшего разбиения на 3 кластера:

# ------------ best clusterization ---------

best\_centroids, best\_centroid\_history, best\_cost, best\_centroid\_indices\_per\_sample = find\_best\_clusterization(x, 3)

print('best cost %s' % best\_cost)

plot\_data\_with\_centroids(x, best\_centroids)

plot\_clusters\_colorized(x, best\_centroid\_indices\_per\_sample, best\_centroids)

plot\_centroids\_trajectory(best\_centroid\_history)

for k in range(3):

plot\_centroid\_trajectory(best\_centroid\_history, k)

# ------------ best clusterization ---------

В результате поиска получена следующее значение минимальной стоимости: 0.5927612624586157.

Покажем и объясним графики, соответствующие лучшему разбиению.

Первый график показывает данные и полученные центроиды (зеленым цветом):

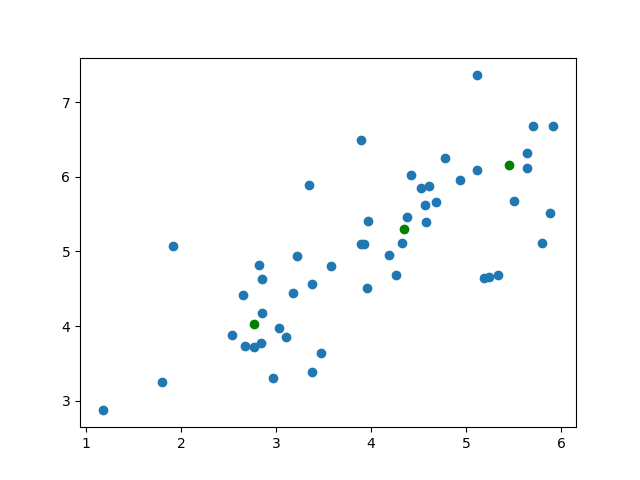


Рисунок 2.8.1. Исходные данные и лучшие центроиды

Далее, покажем график разбиения. На графике все точки разбиты на три группы, группы окрашены разным цветом, а центроиды имеют цвет соответствующего кластера и выделены: изображены большим размером:

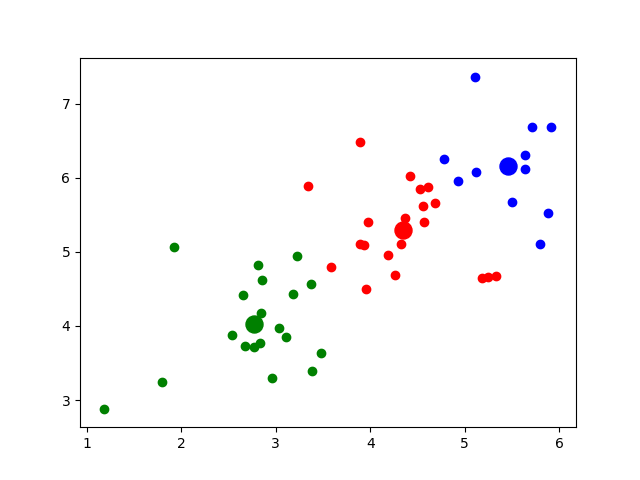


Рисунок 2.8.2. Исходные данные и лучшее разбиение с центроидами

Теперь покажем траекторию движения центроидов. На графике изображены координаты центроидов в ходе алгоритма. Точки, соответствующие координатам центроида в определенной итерации алгоритма, подписаны соответствующим номером:

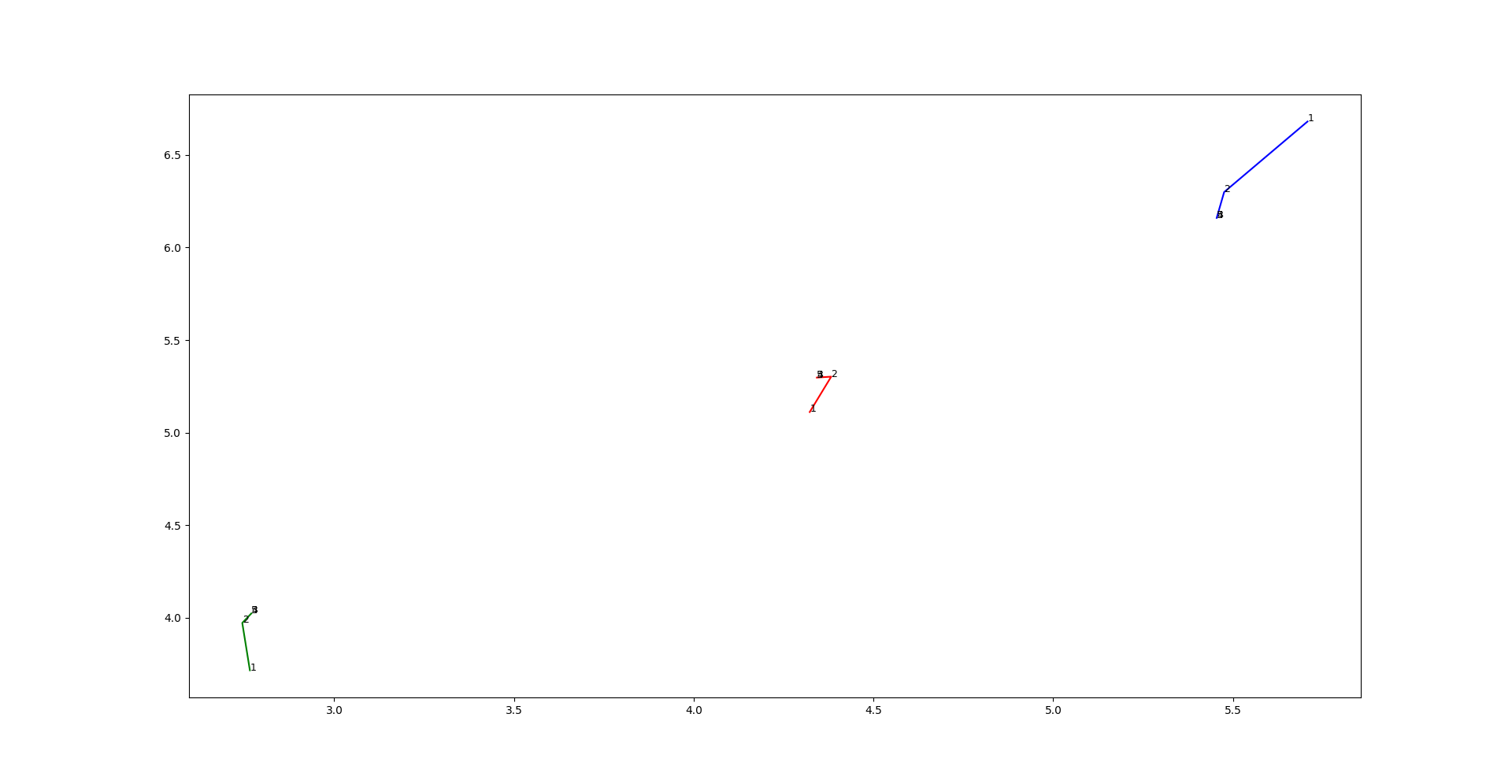


Рисунок 2.8.3. Траектория движения центроида в ходе алгоритма

Изображение довольно мало, поэтому покажем график траектории движения каждого из центроидов в отдельности:

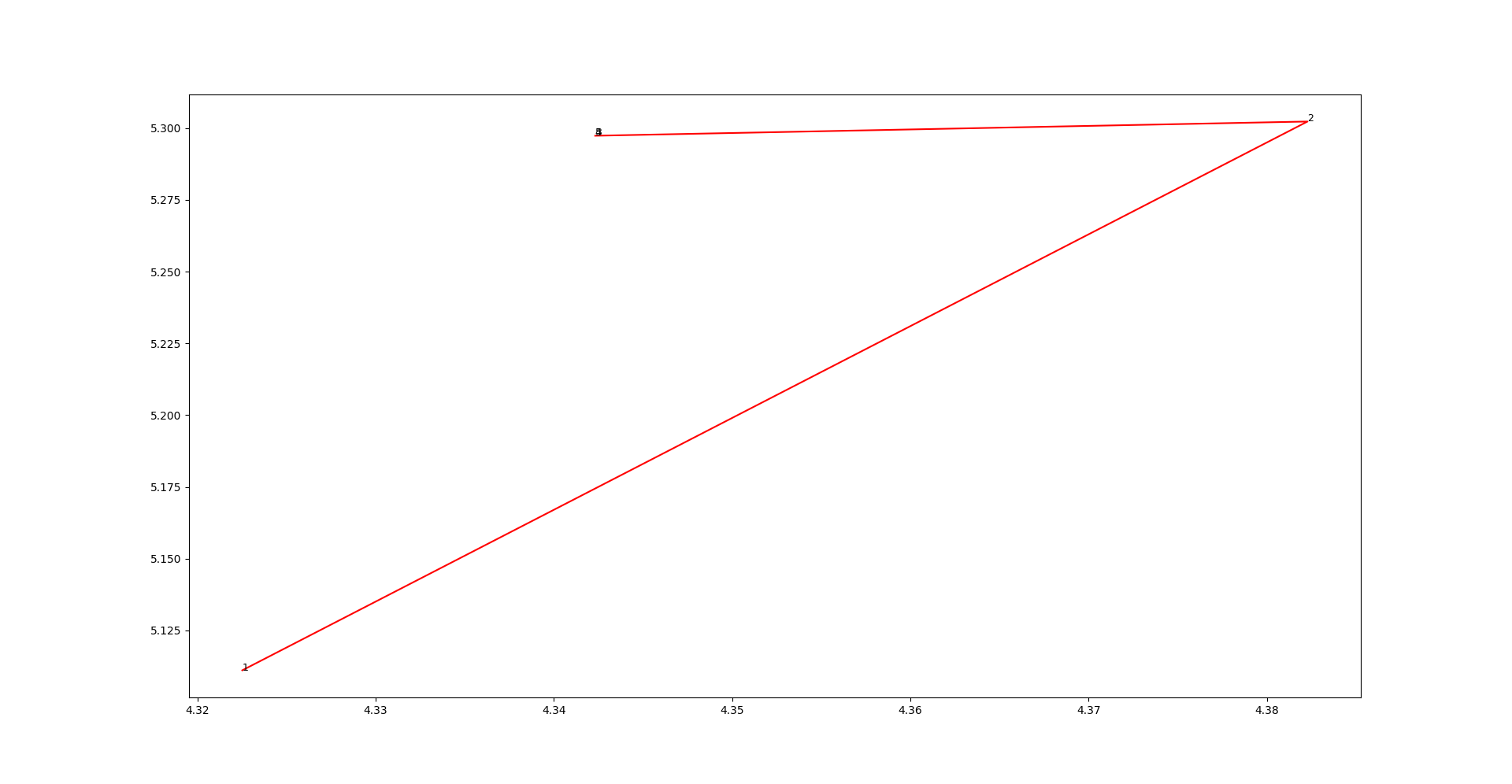


Рисунок 2.8.4. Траектория движения первого центроида

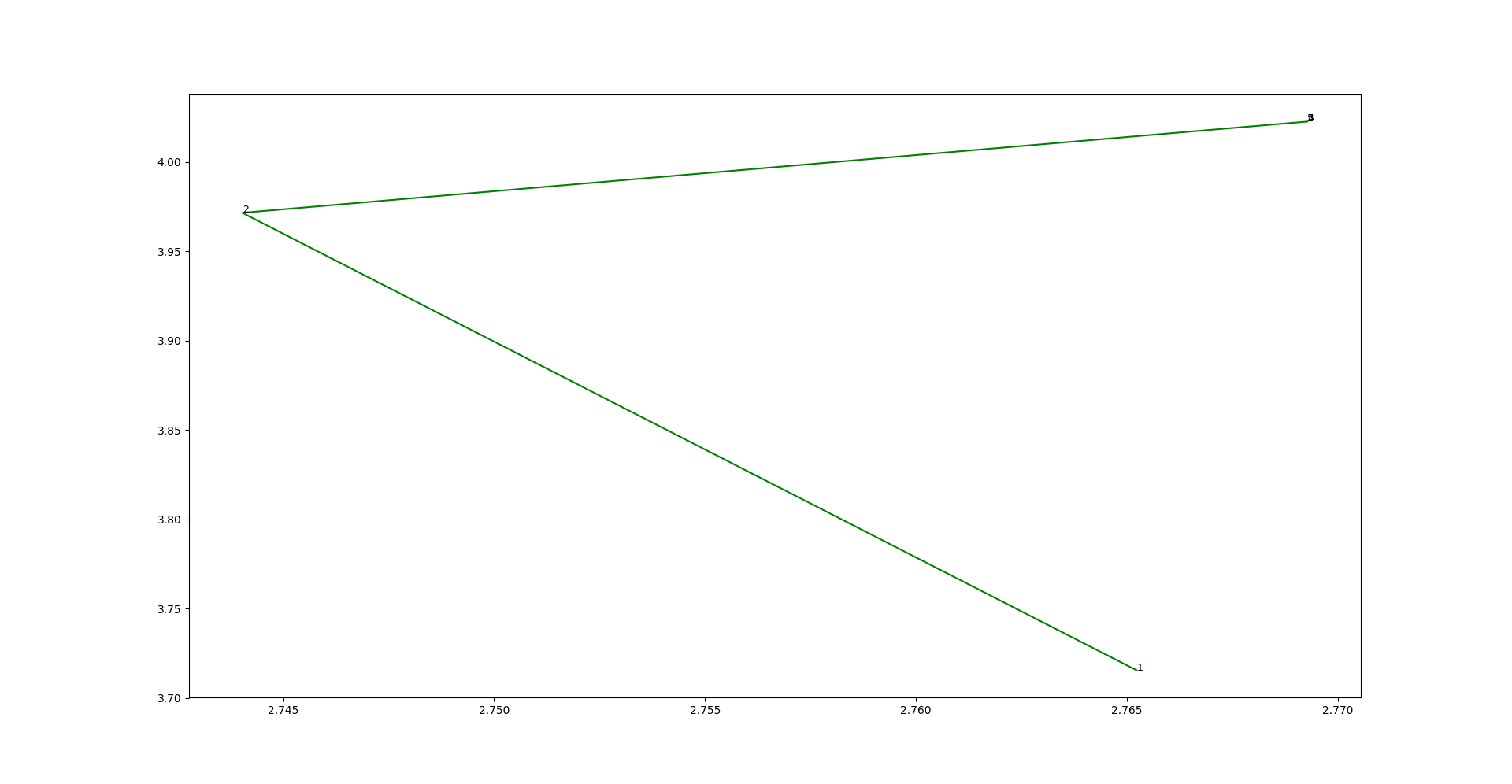


Рисунок 2.8.5. Траектория движения второго центроида

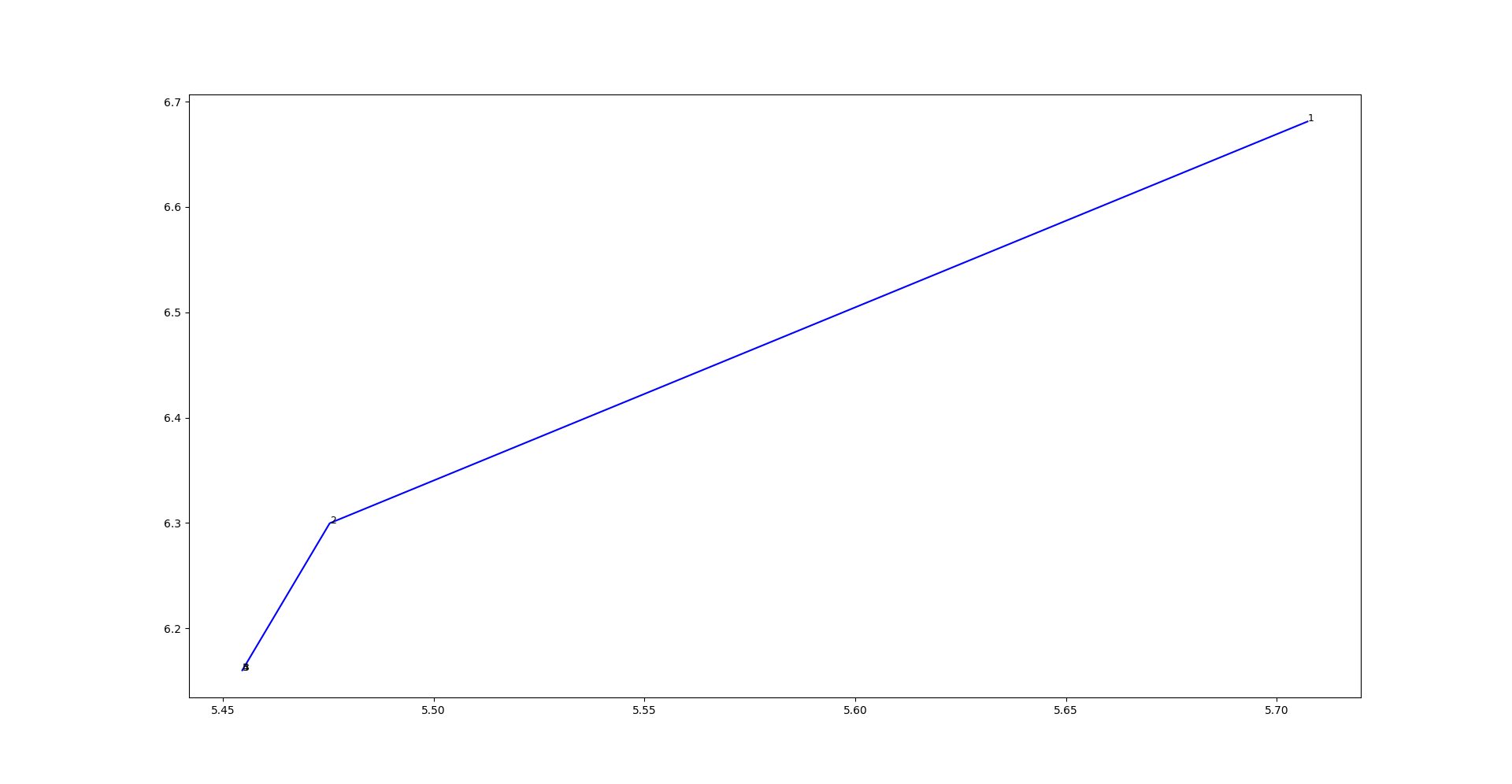


Рисунок 2.8.5. Траектория движения третьего центроида

# 3 Сжатие изображения

### 3.1 Постановка задачи

Дано изображение, требуется сжать его, заменив цвета на 16, полученных разбиением цветов на кластеры.

### 3.2 Чтение данных

Считаем данные:

x = loadmat("bird\_small.mat")['A']

### 3.3 Исходное изображение

Визуализируем исходное изображение:

def plot\_image(x):

plt.imshow(x)

plt.show()

Получившееся изображение:

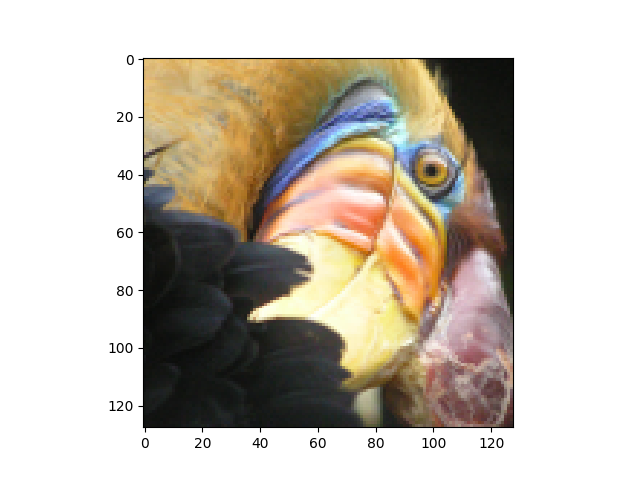


Рисунок 3.3.1. Исходное изображение

### 3.4 Построение сжатия

Для начала, найдем лучшую кластеризацию для изображения, а, точнее, что нас больше интересует, центроиды - 16 цветов, с помощью которых мы будем сжимать изображение.

def build\_compressed\_image(centroids, centroid\_indices\_per\_sample):

image\_size = 128

compressed = np.zeros(shape = (image\_size\*image\_size, 3)).astype(int)

for i in range(len(centroid\_indices\_per\_sample)):

centroid = centroids[centroid\_indices\_per\_sample[i]]

compressed[i] = centroid

return compressed.reshape(image\_size, image\_size, 3)

best\_centroids, best\_cost, best\_centroid\_indices\_per\_sample = find\_best\_clusterization(x.reshape((-1, 3)), 16)

print('best cost %s' % best\_cost)

compressed = build\_compressed\_image(best\_centroids, best\_centroid\_indices\_per\_sample)

plot\_image(compressed)

Здесь мы находим такие 16 цветов и сжимаем изображение следующим образом: цвет каждого пикселя заменяется на цвет центроида кластера, к которому был отнесен цвет данного пикселя во ходе кластеризации. Сам алгоритм кластеризации остался тем же.

### 3.5 Оценка качества сжатия

Покажем результат сжатия изображения:

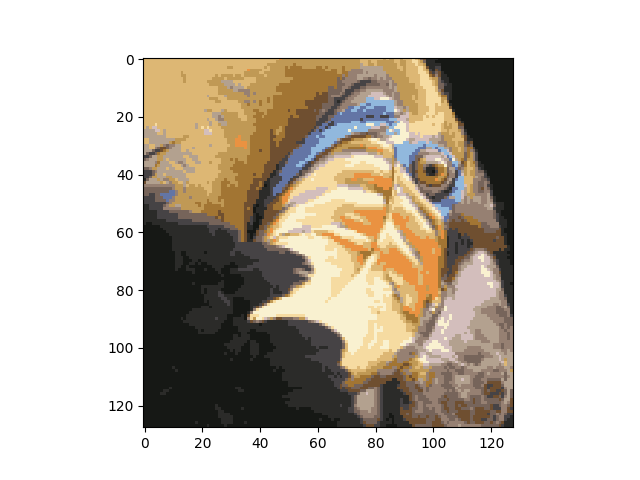


Рисунок 3.5.1. Сжатое изображения

Сжатое изображение занимает 22.8 Кб против 64.5 Кб оригинального. Таким образом, удалось сжать картинку в 2.83 раза.

О качестве можно судить, сравнив оригинальное и сжатое изображение: очевидно, сжатое изображение выглядит на порядок хужи, впрочем, сохранив суть.

### 3.6 Сжатие на другом изображении

Получим результат сжатия следующего изображения (полоса сверху не ошибка алгоритма и т.д., в таком виде изображение найдено в интернете):



Рисунок 3.6.1. Мое изображение

Мое изображение в формате jpg, поэтому, для чистоты эксперимента, для начала изобразим изображение с помощью matplotlib и сохраним его в png:

x = imread('my\_original.jpg')

plot\_image(x)

Полученное изображение занимает 60.6 Кб. Само изображение представлено ниже:

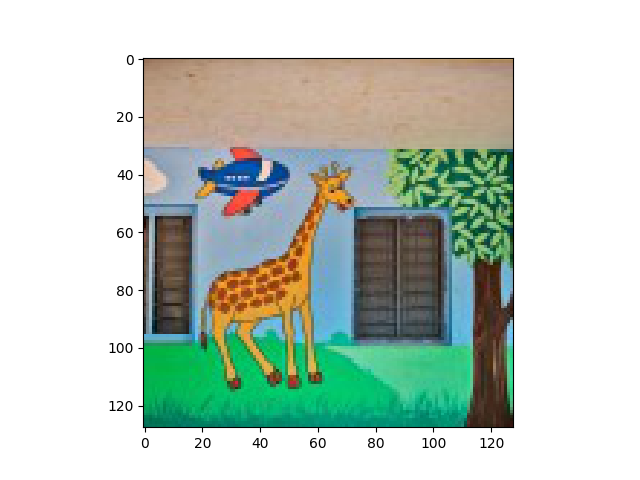


Рисунок 3.6.2. Мое изображение в png

Теперь запустим алгоритм кластеризации:

best\_centroids, best\_cost, best\_centroid\_indices\_per\_sample = find\_best\_clusterization(x.reshape((-1, 3)), 16)

print('best cost %s' % best\_cost)

compressed = build\_compressed\_image(best\_centroids, best\_centroid\_indices\_per\_sample)

plot\_image(compressed)

Результат представлен ниже:

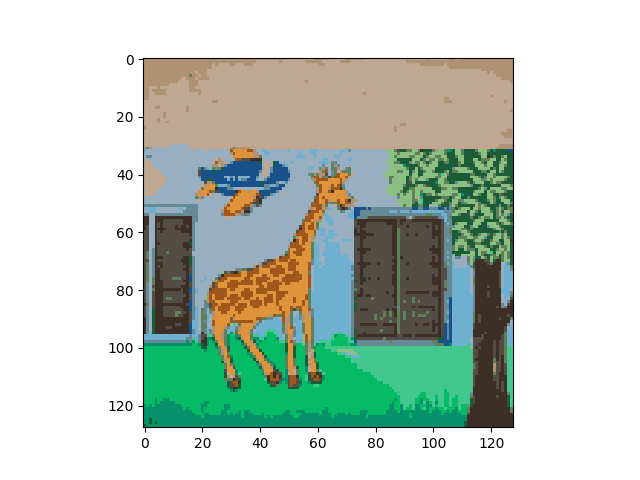


Рисунок 3.6.3. Мое сжатое изображение

Сжатое изображение занимает 19.2 Кб, изображение занимает в 3.16 меньше места.

### 3.7 Иерархическая кластеризация

Алгоритм иерархической кластеризации происходит примерно так: алгоритм начинает работу, разбивая каждый экземпляр выборки на кластер. После этого, алгоритм соединяет ближайшие два кластера в один, пока не образуется один кластер, содержащий все точки. Результаты иерархической кластеризации можно наблюдать, построив дендрограмму, которая выглядит следующим образом:

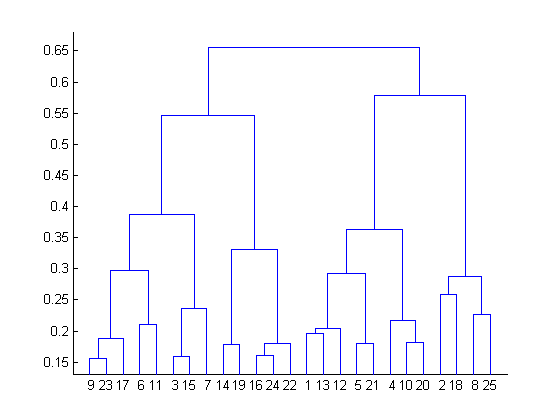


Рисунок 3.7.1. Пример дендрограммы

Дендрограмма позволяет сделать выбор количества кластеров осознанно, а не с помощью переборов различных значений количества кластеров и визуализации получившегося разбиения (не существует способа “идеально” выбрать количество кластеров, все упирается в business value, которое в конечном счете хотят получить из разбиения).

Мы же ограничимся сжатием алгоритмом иерархической кластеризации с 16 кластерами:

K = 16

x = imread('my\_original.jpg').reshape(-1, 3)

cluster = AgglomerativeClustering(n\_clusters=K, affinity='euclidean', linkage='ward')

cluster.fit\_predict(x)

centroid\_indices\_per\_sample = cluster.labels\_

centroids = compute\_centroids(x, centroid\_indices\_per\_sample, K)

compressed = build\_compressed\_image(centroids, centroid\_indices\_per\_sample)

plot\_image(compressed)

Использована библиотечная реализация иерархической кластеризации AgglomerativeClustering. Параметр affinity = ‘euclidean’ указывает, как рассчитывать связь между кластерами (linkage). Параметр linkage = ‘ward’ конфигурирует алгоритм уменьшать дисперсию между значениями этой связи (путем соединения кластеров).

Полученное изображение можно видеть ниже:

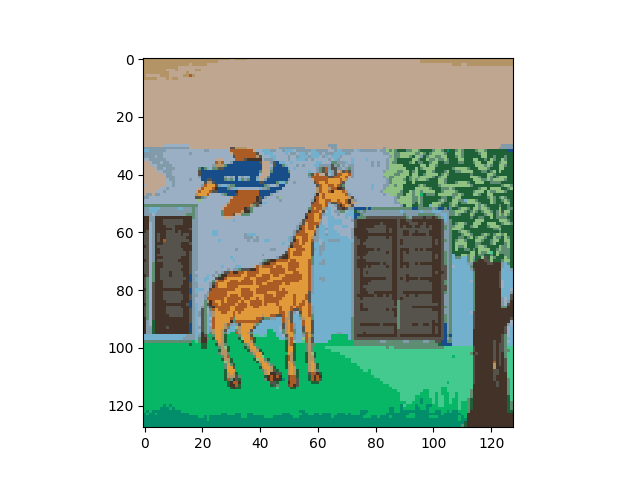


Рисунок 3.7.2. Результат работы иерархического алгоритма кластеризации

В нашем случае, результат примерно такой же, как и в k-means: заметно сжатое изображение. Занимает оно примерно столько же места: 18.2 Кб, что предсказуемо.

Единственная разница, которую можно видеть (возможно, имеет место быть некоторое субъективное мнение), что на изображении, сжатом иерархической кластеризацией, больше темных цветом (темно-коричневый, оранжевый), тогда как в изображении, сжатом k-means, некоторые такие цвета светлее (оранжевый просто желтый). Также, опять же субъективно, на изображении, сжатом иерархической кластеризацией, меньше “неправильных” цветов пикселей: цветов, которые не подходят на части этого изображения по смыслу. Например, на изображении k-means присутствует зеленый цвет на темно-коричневой двери, чего почти нет на другом сжатом изображении.