Лабораторная работа №8

# Карта глубины из стереопары

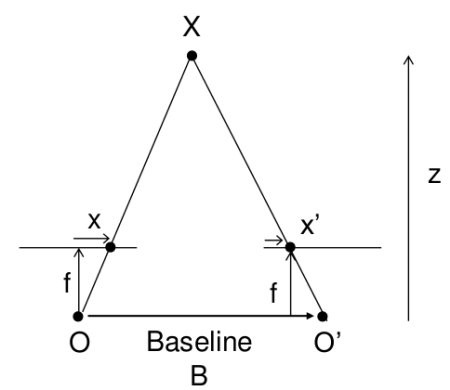
*Цель*

*В этой сессии,*

*Мы научимся создавать карту глубины из стереоизображений.*

Основы

На прошлой сессии мы рассмотрели основные понятия, такие как эпиполярные ограничения и другие связанные с ними термины. Мы также видели, что если у нас есть два изображения одной и той же сцены, мы можем получить информацию о глубине интуитивно. Ниже приводится изображение и несколько простых математических формул, которые доказывают эту интуицию.



Приведенная выше диаграмма содержит эквивалентные треугольники. Написание их эквивалентных уравнений даст нам следующий результат:

disparity = x - x' = \frac{Bf}{Z}

x и x' - это расстояние между точками в плоскости изображения, соответствующими точке сцены point3D, и их центром камеры. B-это расстояние между двумя камерами (которое мы знаем), а f-фокусное расстояние камеры (уже известное). Короче говоря, приведенное выше уравнение говорит о том, что глубина точки в сцене обратно пропорциональна разнице расстояний между соответствующими точками изображения и их центрами камер. Таким образом, с помощью этой информации мы можем получить глубину всех пикселей в изображении.

Таким образом, он находит соответствующие соответствия между двумя изображениями. Мы уже видели, как ограничение эпилина делает эту операцию более быстрой и точной. Как только он находит совпадения, он находит несоответствие. Давайте посмотрим, как мы можем сделать это с OpenCV.

**# import OpenCV and pyplot**

**import cv2 as cv**

**from matplotlib import pyplot as plt**

**# read left and right images**

**imgR = cv.imread('800xdsc00645\_l.jpg', 0)**

**imgL = cv.imread('800xdsc00645\_r.jpg', 0)**

**# creates StereoBm object**

**stereo = cv.StereoBM\_create(numDisparities = 16,**

**blockSize = 15)**

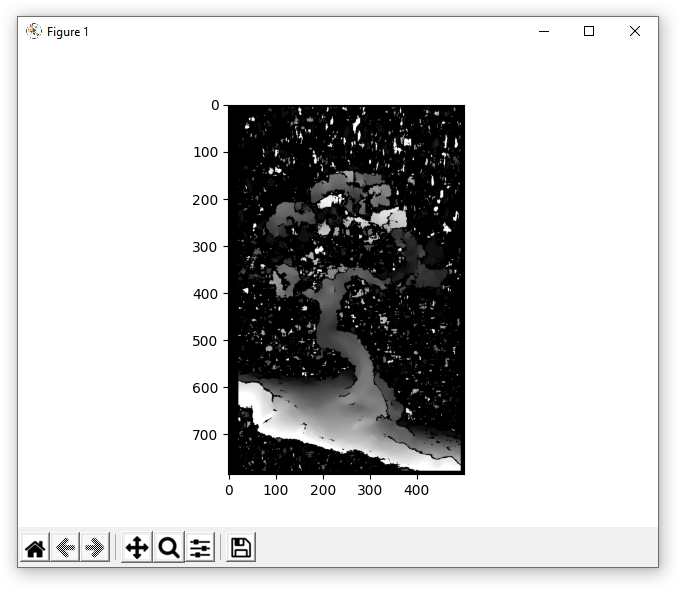
**# computes disparity**

**disparity = stereo.compute(imgL, imgR)**

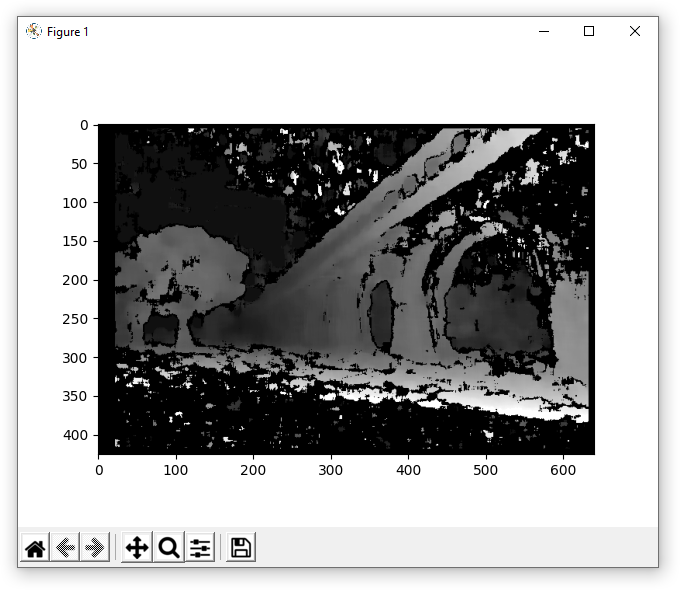
**# displays image as grayscale and plotted**

**plt.imshow(disparity, 'gray')**

**plt.show()**

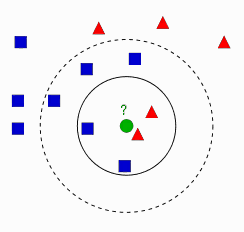




# K-Nearest Neighbour

## Understanding k-Nearest Neighbour

kNN - один из простейших алгоритмов классификации, доступных для обучения под наблюдением. Идея заключается в поиске наиболее близкого соответствия данных теста в пространстве характеристик.



На изображении есть два семейства, синие квадраты и красные треугольники. Мы называем каждую семью классом. Их дома показаны на карте города, которую мы называем пространством объектов. (Пространство объектов можно рассматривать как пространство, в котором проецируются все данные. Например, рассмотрим двумерное координатное пространство. Все данные имеют два объекта, координаты x и Y. Вы можете представить эти данные в своем двумерном координатном пространстве, не так ли? Теперь представьте, что если есть три объекта, вам нужно 3D-пространство. Теперь рассмотрим N объектов, где вам нужно N-мерное пространство, верно? Это N-мерное пространство является его пространством признаков. На нашем изображении вы можете рассматривать его как 2D-случай с двумя функциями).

Теперь новый член приходит в город и создает новый дом, который показан как зеленый круг. Он должен быть добавлен в одну из этих сине-красных семей. Мы называем этот процесс классификацией. Что же нам делать? Поскольку мы имеем дело с kNN, давайте применим этот алгоритм.

Один из способов-проверить, кто его ближайший сосед. Из изображения ясно, что это семейство красных треугольников. Таким образом, он также добавляется в Красный треугольник. Этот метод называется просто ближайший сосед, потому что классификация зависит только от ближайшего соседа.

Но тут есть одна проблема. Красный треугольник может быть самым близким. Но что, если рядом с ним много синих квадратов? Тогда синие квадраты имеют больше силы в этой местности, чем Красный треугольник. Так что просто проверить ближайший из них недостаточно. Вместо этого мы проверяем некоторые k ближайших семей. Тогда, кто бы ни был в них большинством, новый парень принадлежит к этой семье. В нашем образе возьмем k=3, то есть 3 ближайших семейства. У него есть два красных и один синий (есть два синих равноудаленных, но так как k=3, мы берем только один из них), поэтому его снова следует добавить к Красному семейству. Но что, если мы возьмем k=7? Тогда у него есть 5 синих семейств и 2 красных семейства. Отлично!! Теперь он должен быть добавлен в Голубую семью. Так что все это меняется со значением k. Еще более забавно, что если k = 4? У него есть 2 красных и 2 синих соседа. Это галстук !!! Поэтому лучше взять k как нечетное число. Поэтому этот метод называется k-ближайшим соседом, поскольку классификация зависит от k ближайших соседей.

Опять же, в kNN мы действительно рассматриваем k соседей, но мы придаем равное значение всем, верно? Разве это справедливость? Например, возьмем случай k=4. Мы сказали, что это галстук. Но видите ли, две красные семьи ближе к нему, чем две другие синие семьи. Так что он имеет больше прав быть добавленным к Красному. Так как же мы можем математически объяснить это? Мы даем несколько весов каждой семье в зависимости от их расстояния до новичка. Ибо те, кто рядом с ним, получают более высокие веса, в то время как те, кто далеко, получают более низкие веса. Затем мы складываем суммарные веса каждой семьи отдельно. Тот, кто получает наибольший общий вес, новичок идет в эту семью. Это называется **модифицированным kNN**.

Итак, какие важные вещи вы видите здесь?

Тебе ведь нужна информация обо всех домах в городе, верно? Потому что мы должны проверить расстояние от новичка до всех существующих домов, чтобы найти ближайшего соседа. Если есть много домов и семей, это занимает много памяти, а также больше времени для расчета.

Существует почти нулевое время для любого вида обучения или подготовки.

## kNN in OpenCV

Мы сделаем здесь простой пример с двумя семействами (классами), как и выше. Тогда в следующей главе мы сделаем гораздо более наглядный пример.

Итак, здесь мы обозначаем Красное семейство как класс-0 (обозначается 0), а синее семейство как класс-1 (обозначается 1). Мы создаем 25 семейств или 25 обучающих данных и обозначаем их либо классом-0, либо классом-1. Все это мы делаем с помощью генератора случайных чисел в Numpy.

Затем мы строим его с помощью Matplotlib. Красные семейства показаны в виде красных треугольников, а синие-в виде синих квадратов.

**import cv2**

**import numpy as np**

**import matplotlib.pyplot as plt**

**# Feature set containing (x,y) values of 25 known/training data**

**trainData = np.random.randint(0,100,(25,2)).astype(np.float32)**

**# Labels each one either Red or Blue with numbers 0 and 1**

**responses = np.random.randint(0,2,(25,1)).astype(np.float32)**

**# Take Red families and plot them**

**red = trainData[responses.ravel()==0]**

**plt.scatter(red[:,0],red[:,1],80,'r','^')**

**# Take Blue families and plot them**

**blue = trainData[responses.ravel()==1]**

**plt.scatter(blue[:,0],blue[:,1],80,'b','s')**

**plt.show()**

вы получите нечто похожее на наше первое изображение. Поскольку вы используете генератор случайных чисел, вы будете получать разные данные каждый раз, когда вы запускаете код.

Затем инициируйте алгоритм kNN и передайте trainData и ответы для обучения kNN (он строит дерево поиска).

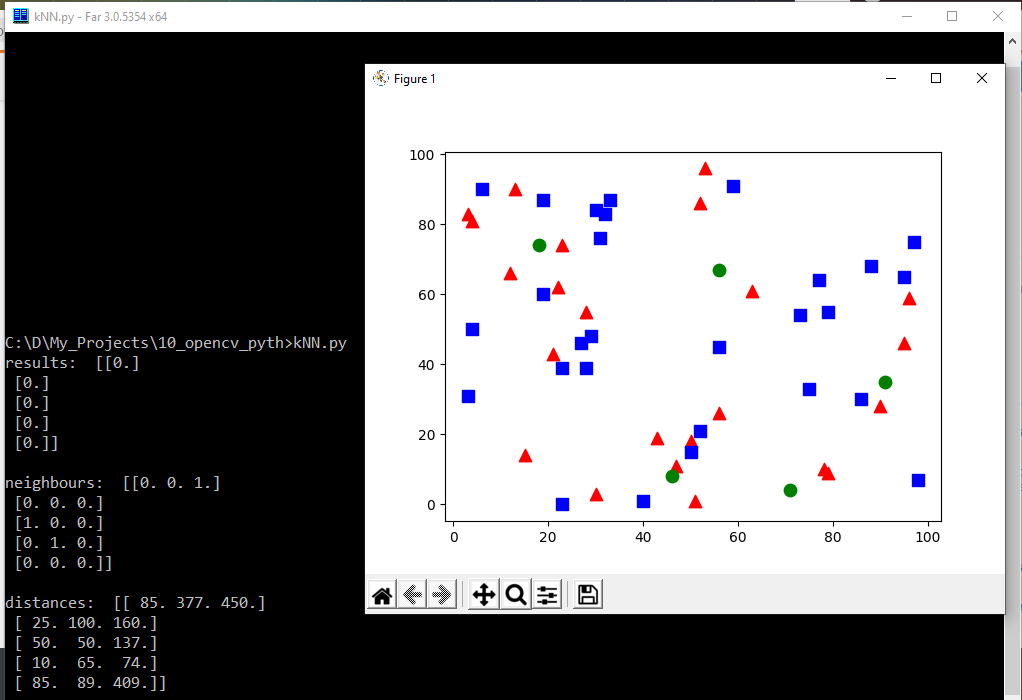
Затем мы приведем одного новичка и классифицируем его в семью с помощью kNN в OpenCV. Прежде чем идти в kNN, нам нужно кое-что узнать о наших тестовых данных (данные новичков). Наши данные должны быть плавающий массив точке С число размер \; В \; ними \номер \; в \; особенности. Затем мы находим ближайших соседей новоприбывшего. Мы можем указать, сколько соседей мы хотим. Он возвращает:

- Ярлык, данный новичку в зависимости от теории КНН, которую мы видели ранее. Если вам нужен алгоритм ближайшего соседа, просто укажите k=1, где k - количество соседей.

- Метки k-ближайших соседей.

- Соответствующие расстояния от новичка до каждого ближайшего соседа.

Так что давайте посмотрим, как это работает. Новый посетитель отмечен зеленым цветом.



**import cv2**

**import numpy as np**

**import matplotlib.pyplot as plt**

**trainData = np.random.randint(0,100,(51,2)).astype(np.float32)**

**responses = np.random.randint(0,2,(51,1)).astype(np.float32)**

**red = trainData[responses.ravel()==0]**

**plt.scatter(red[:,0],red[:,1],80,'r','^')**

**blue = trainData[responses.ravel()==1]**

**plt.scatter(blue[:,0],blue[:,1],80,'b','s')**

**newcomer = np.random.randint(0,100,(5,2)).astype(np.float32)**

**plt.scatter(newcomer[:,0],newcomer[:,1],80,'g','o')**

**knn = cv2.ml.KNearest\_create()**

**knn.train(trainData,cv2.ml.ROW\_SAMPLE,responses)**

**ret, results, neighbours, dist = knn.findNearest(newcomer, 3)**

**print ("results: ", results,"\n")**

**print ("neighbours: ", neighbours,"\n")**

**print ("distances: ", dist)**

**plt.show()**

## Распознавание рукописного текста с kNN

Наша цель-создать приложение, которое может считывать рукописные цифры. Для этого нам нужны некоторые обучающие данные и некоторые тестовые данные. OpenCV поставляется с изображением digits.png (в папке opencv/samples/data/), которое содержит 5000 рукописных цифр (по 500 для каждой цифры). Каждая цифра представляет собой изображение 20х20. Итак, наш первый шаг состоит в том, чтобы разделить это изображение на 5000 различных цифровых изображений. Затем для каждой цифры (изображение 20х20) мы сглаживаем ее в одну строку с 400 пикселями. Это наш набор признаков, то есть значения интенсивности всех пикселей. Это самый простой набор функций, который мы можем создать. Мы используем первые 250 выборок каждой цифры в качестве обучающих данных, а остальные 250 выборок-в качестве тестовых данных. Так что давайте сначала подготовим их.

**import numpy as np**

**import cv2 as cv**

**img = cv.imread('digits.png')**

**gray = cv.cvtColor(img,cv.COLOR\_BGR2GRAY)**

**# Now we split the image to 5000 cells, each 20x20 size**

**cells = [np.hsplit(row,100) for row in np.vsplit(gray,50)]**

**# Make it into a Numpy array: its size will be (50,100,20,20)**

**x = np.array(cells)**

**# Now we prepare the training data and test data**

**train = x[:,:50].reshape(-1,400).astype(np.float32) # Size = (2500,400)**

**test = x[:,50:100].reshape(-1,400).astype(np.float32) # Size = (2500,400)**

**# Create labels for train and test data**

**k = np.arange(10)**

**train\_labels = np.repeat(k,250)[:,np.newaxis]**

**test\_labels = train\_labels.copy()**

**# Initiate kNN, train it on the training data, then test it with the test data with k=1**

**knn = cv.ml.KNearest\_create()**

**knn.train(train, cv.ml.ROW\_SAMPLE, train\_labels)**

**ret,result,neighbours,dist = knn.findNearest(test,k=5)**

**# Now we check the accuracy of classification**

**# For that, compare the result with test\_labels and check which are wrong**

**matches = result==test\_labels**

**correct = np.count\_nonzero(matches)**

**accuracy = correct\*100.0/result.size**

**print( accuracy )**

Итак, наше базовое приложение OCR готово. Этот конкретный пример дал точность 91%. Один из вариантов повышения точности-добавить больше данных для обучения, особенно для цифр, где у нас было больше ошибок.

Вместо того чтобы находить эти обучающие данные каждый раз, когда я запускаю приложение, я лучше сохраню их, чтобы в следующий раз я мог непосредственно прочитать эти данные из файла и начать классификацию. Это можно сделать с помощью некоторых функций Numpy, таких как np.savetxt, np.savez, np.load и т. д. Пожалуйста, проверьте документы NumPy для получения более подробной информации.

**# Save the data**

**np.savez('knn\_data.npz',train=train, train\_labels=train\_labels)**

**# Now load the data**

**with np.load('knn\_data.npz') as data:**

**print( data.files )**

**train = data['train']**

**train\_labels = data['train\_labels']**

В моей системе он занимает около 4,4 МБ памяти. Поскольку мы используем значения интенсивности (данные uint8) в качестве объектов, было бы лучше сначала преобразовать данные в np.uint8, а затем сохранить их. В этом случае он занимает всего 1,1 МБ. Затем во время загрузки вы можете конвертировать обратно в float32.

## Распознавание символов английского алфавита

Далее мы сделаем то же самое для английского алфавита, но есть небольшое изменение в данных и наборе функций. Здесь вместо изображений OpenCV поставляется с файлом данных, распознаванием букв.данные в папке opencv/samples/cpp/. Если вы откроете его, то увидите 20000 строк, которые на первый взгляд могут показаться мусором. На самом деле, в каждой строке первый столбец-это буква, которая является нашей меткой. Следующие 16 чисел, следующие за ним, - это различные функции. Эти функции получены из репозитория машинного обучения UCI [].

Доступно 20000 образцов, поэтому мы берем первые 10000 в качестве учебных образцов, а остальные 10000-в качестве тестовых образцов. Мы должны изменить буквы на символы ascii, потому что мы не можем работать с буквами напрямую.

**import cv2 as cv**

**import numpy as np**

**# Load the data and convert the letters to numbers**

**data= np.loadtxt('letter-recognition.data', dtype= 'float32', delimiter = ',',**

**converters= {0: lambda ch: ord(ch)-ord('A')})**

**# Split the dataset in two, with 10000 samples each for training and test sets**

**train, test = np.vsplit(data,2)**

**# Split trainData and testData into features and responses**

**responses, trainData = np.hsplit(train,[1])**

**labels, testData = np.hsplit(test,[1])**

**# Initiate the kNN, classify, measure accuracy**

**knn = cv.ml.KNearest\_create()**

**knn.train(trainData, cv.ml.ROW\_SAMPLE, responses)**

**ret, result, neighbours, dist = knn.findNearest(testData, k=5)**

**correct = np.count\_nonzero(result == labels)**

**accuracy = correct\*100.0/10000**

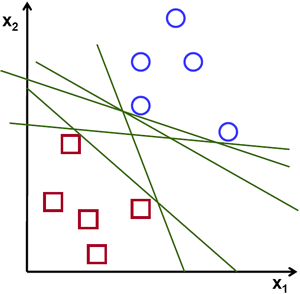
**print( accuracy )**

Здесь мы использовали k=5. Что произойдет, если вы попробуете другие значения k? Можете ли вы найти значение, которое максимизирует точность (минимизирует количество ошибок)?

# SVM

## Теория - Линейно Разделяемые Данные

Рассмотрим изображение ниже, которое имеет два типа данных, красный и синий. В kNN для получения тестовых данных мы измеряли его расстояние до всех обучающих выборок и брали ту, которая имела минимальное расстояние. Требуется много времени для измерения всех расстояний и много памяти для хранения всех обучающих выборок. Но учитывая данные, приведенные в изображении, нужно ли нам так много?



## Распознавание рукописных цифр SVM

В kNN, мы использовали интенсивность пикселя в качестве вектора признаков. На этот раз мы будем использовать гистограмму ориентированных градиентов (HOG) в качестве векторов признаков.

Здесь, прежде чем найти HOG, мы снимаем изображение, используя его моменты второго порядка. Итак, вначале определимся с функцией deskew(), которая принимает изображения и выравнивает его:

**def deskew(img):**

**m = cv.moments(img)**

**if abs(m['mu02']) < 1e-2:**

**return img.copy()**

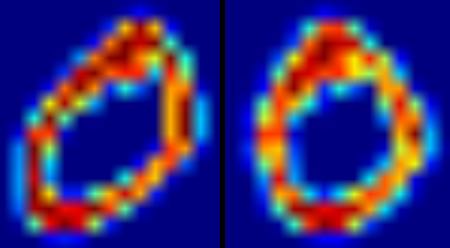
**skew = m['mu11']/m['mu02']**

**M = np.float32([[1, skew, -0.5\*SZ\*skew], [0, 1, 0]])**

**img = cv.warpAffine(img,M,(SZ, SZ),flags=affine\_flags)**

**return img**

До и после:



Далее мы должны найти HOG-дескриптор для каждой ячейки. Для этого мы находим производные Собеля каждой ячейки в направлении X и Y. Затем найдите их величину и направление градиента в каждом пикселе. Этот градиент квантован до 16 целых значений. Разделите это изображение на четыре подквадрата. Для каждого субквадрата вычислите гистограмму направления (16 ячеек), взвешенную по их величине. Таким образом, каждый субквадрат дает вам вектор, содержащий 16 значений. Четыре таких вектора (из четырех подквадратов) вместе дают нам вектор признаков, содержащий 64 значения. Это вектор признаков, который мы используем для обучения наших данных.

**def hog(img):**

**gx = cv.Sobel(img, cv.CV\_32F, 1, 0)**

**gy = cv.Sobel(img, cv.CV\_32F, 0, 1)**

**mag, ang = cv.cartToPolar(gx, gy)**

**bins = np.int32(bin\_n\*ang/(2\*np.pi)) # quantizing binvalues in (0...16)**

**bin\_cells = bins[:10,:10], bins[10:,:10], bins[:10,10:], bins[10:,10:]**

**mag\_cells = mag[:10,:10], mag[10:,:10], mag[:10,10:], mag[10:,10:]**

**hists = [np.bincount(b.ravel(), m.ravel(), bin\_n) for b, m in zip(bin\_cells, mag\_cells)]**

**hist = np.hstack(hists) # hist is a 64 bit vector**

**return hist**

Наконец, как и в предыдущем случае, мы начинаем с разбиения нашего большого набора данных на отдельные ячейки. Для каждой цифры, 250 ячеек зарезервированы для обучающих данных и остальные 250 данных, предназначенных для проведения испытаний.

**import cv2 as cv**

**import numpy as np**

**SZ=20**

**bin\_n = 16 # Number of bins**

**affine\_flags = cv.WARP\_INVERSE\_MAP|cv.INTER\_LINEAR**

**def deskew(img):**

**m = cv.moments(img)**

**if abs(m['mu02']) < 1e-2:**

**return img.copy()**

**skew = m['mu11']/m['mu02']**

**M = np.float32([[1, skew, -0.5\*SZ\*skew], [0, 1, 0]])**

**img = cv.warpAffine(img,M,(SZ, SZ),flags=affine\_flags)**

**return img**

**def hog(img):**

**gx = cv.Sobel(img, cv.CV\_32F, 1, 0)**

**gy = cv.Sobel(img, cv.CV\_32F, 0, 1)**

**mag, ang = cv.cartToPolar(gx, gy)**

**bins = np.int32(bin\_n\*ang/(2\*np.pi)) # quantizing binvalues in (0...16)**

**bin\_cells = bins[:10,:10], bins[10:,:10], bins[:10,10:], bins[10:,10:]**

**mag\_cells = mag[:10,:10], mag[10:,:10], mag[:10,10:], mag[10:,10:]**

**hists = [np.bincount(b.ravel(), m.ravel(), bin\_n) for b, m in zip(bin\_cells, mag\_cells)]**

**hist = np.hstack(hists) # hist is a 64 bit vector**

**return hist**

**img = cv.imread('digits.png',0)**

**if img is None:**

**raise Exception("we need the digits.png image from samples/data here !")**

**cells = [np.hsplit(row,100) for row in np.vsplit(img,50)]**

**# First half is trainData, remaining is testData**

**train\_cells = [ i[:50] for i in cells ]**

**test\_cells = [ i[50:] for i in cells]**

**deskewed = [list(map(deskew,row)) for row in train\_cells]**

**hogdata = [list(map(hog,row)) for row in deskewed]**

**trainData = np.float32(hogdata).reshape(-1,64)**

**responses = np.repeat(np.arange(10),250)[:,np.newaxis]**

**svm = cv.ml.SVM\_create()**

**svm.setKernel(cv.ml.SVM\_LINEAR)**

**svm.setType(cv.ml.SVM\_C\_SVC)**

**svm.setC(2.67)**

**svm.setGamma(5.383)**

**svm.train(trainData, cv.ml.ROW\_SAMPLE, responses)**

**svm.save('svm\_data.dat')**

**deskewed = [list(map(deskew,row)) for row in test\_cells]**

**hogdata = [list(map(hog,row)) for row in deskewed]**

**testData = np.float32(hogdata).reshape(-1,bin\_n\*4)**

**result = svm.predict(testData)[1]**

**mask = result==responses**

**correct = np.count\_nonzero(mask)**

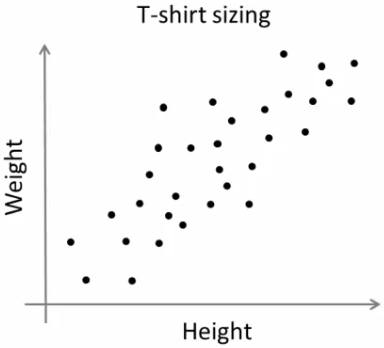
**print(correct\*100.0/result.size)**

# K-Means Clustering

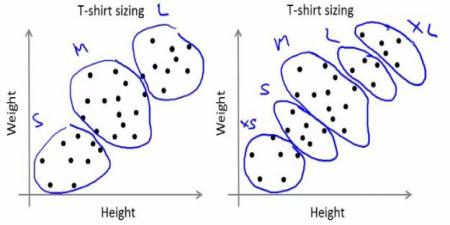
## Understanding K-Means Clustering

Проблема с размером футболки

Рассмотрим компанию, которая собирается выпустить на рынок новую модель футболки. Очевидно, что им придется производить модели разных размеров, чтобы удовлетворить людей всех размеров. Поэтому компания делает данные о росте и весе людей и выводит их на график, как показано ниже:



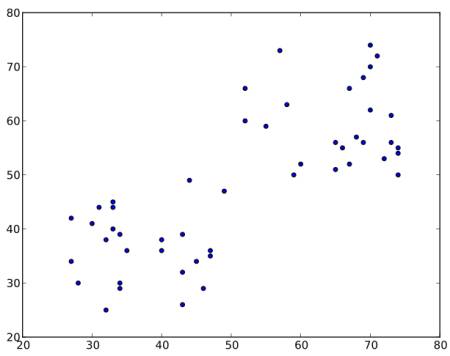
Компания не может создавать футболки всех размеров. Вместо этого они делят людей на мелких, средних и крупных и производят только эти 3 модели, которые будут соответствовать всем людям. Эта группировка людей в три группы может быть выполнена с помощью кластеризации k-средних, и алгоритм предоставляет нам лучшие 3 размера, которые удовлетворят всех людей. А если это не так, компания может разделить людей на несколько групп, может быть, пять, и так далее. Проверьте изображение ниже :



Как это работает ?

Этот алгоритм является итеративным процессом. Мы объясним это шаг за шагом с помощью образов.

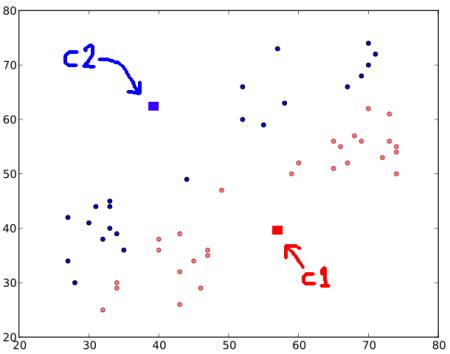
Рассмотрим набор данных, как показано ниже ( вы можете рассматривать его как проблему футболки). Нам нужно сгруппировать эти данные в две группы.



Шаг : 1 - алгоритм случайным образом выбирает два центроида, C1 и C2 (иногда в качестве центроидов берутся любые два данных).

Шаг : 2 - он вычисляет расстояние от каждой точки до обоих центроидов. Если тестовые данные находятся ближе к C1, то эти данные помечаются знаком "0". Если он ближе к С2, то помечается как " 1 "(если там больше центроидов, то помечается как "2","3" и т. д.).

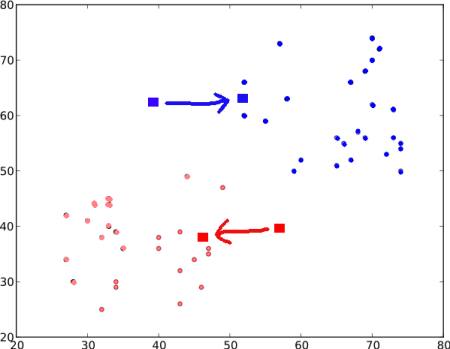
В нашем случае мы будем окрашивать все "0", помеченные красным цветом, и "1", помеченные синим цветом. Таким образом, мы получаем следующее изображение после вышеуказанных операций



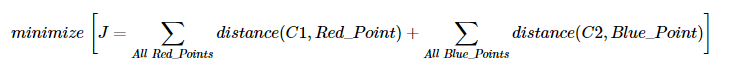
Шаг : 3 - Далее мы вычисляем среднее значение всех синих точек и красных точек отдельно, и это будут наши новые центроиды. То есть С1 и С2 смещаются на вновь вычисленные центроиды. (Помните, что показанные изображения не являются истинными значениями и не соответствуют истинному масштабу, это только для демонстрации).

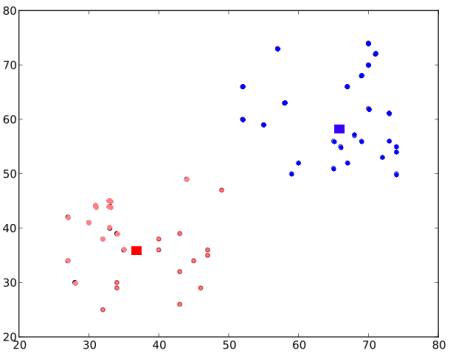
И снова выполните шаг 2 с новыми центроидами и обозначьте данные как " 0 " и "1".

Таким образом, мы получаем результат, как показано ниже :



Теперь Шаг 2 и Шаг 3 повторяются до тех пор, пока оба центроида не сойдутся в неподвижные точки. \*(Или он может быть остановлен в зависимости от критериев, которые мы предоставляем, например, максимальное количество итераций, или достигается определенная точность и т. д.)\* Эти точки таковы, что сумма расстояний между тестовыми данными и их соответствующими центроидами минимальна. Или просто сумма расстояний между C1 ↔ Red\_Points и C2 ↔ Blue\_Points минимальна.





Так что это всего лишь интуитивное понимание кластеризации K-средних. Для получения более подробной информации и математического объяснения, пожалуйста, прочтите любые стандартные учебники машинного обучения или проверьте ссылки в дополнительных ресурсах. Это просто верхний слой кластеризации K-средних. Существует множество модификаций этого алгоритма, например, как выбрать начальные центроиды, как ускорить итерационный процесс и т. д.

## K-Means Clustering in OpenCV

Научитесь использовать функцию **cv.kmeans()** в OpenCV для кластеризации данных

Параметры:

**samples**: он должен иметь тип данных np.float32, и каждый объект должен быть помещен в один столбец.

**nclusters(K)** : количество кластеров, требуемых в конечных

**criteria**: это критерий завершения итерации. Когда этот критерий удовлетворен, итерация алгоритма останавливается. На самом деле это должен быть кортеж из 3 параметров. Это `( type, max\_iter, epsilon )`:

тип критериев прекращения действия договора. Он имеет 3 флага, как показано ниже:

- cv.TERM\_CRITERIA\_EPS - остановить итерацию алгоритма, если достигнута заданная точность, Эпсилон.

- cv.TERM\_CRITERIA\_MAX\_ITER - остановить алгоритм после указанного количества итераций, max\_iter.

- cv.TERM\_CRITERIA\_EPS + cv.TERM\_CRITERIA\_MAX\_ITER - остановка итерации при выполнении любого из вышеперечисленных условий.

- max\_iter - целое число, задающее максимальное число итераций.

- epsilon - требуемая точность

**attempts** : флаг для указания количества раз, когда алгоритм выполняется с использованием различных начальных меток. Алгоритм возвращает метки, которые дают наилучшую компактность. Эта компактность возвращается в качестве вывода.

**flags**: этот флаг используется для указания того, как берутся начальные центры. Обычно для этого используются два флага : cv.KMEANS\_PP\_CENTERS и cv.KMEANS\_RANDOM\_CENTERS.

Выходные параметры:

- compactness (компактность) : это сумма квадратов расстояний от каждой точки до их соответствующих центров.

- labels (метки): это массив меток (такой же, как " код "в предыдущей статье), где каждый элемент помечен "0", "1".....

- centers (центры): это массив центров кластеров.

### 1. Данные только с одной функцией

Считайте, что у вас есть набор данных только с одним признаком, то есть одномерный. Например, мы можем взять нашу проблему с футболкой, где вы используете только рост людей, чтобы решить размер футболки.

Итак, мы начинаем с создания данных и построения их в Matplotlib

**import numpy as np**

**import cv2 as cv**

**from matplotlib import pyplot as plt**

**x = np.random.randint(25,100,25)**

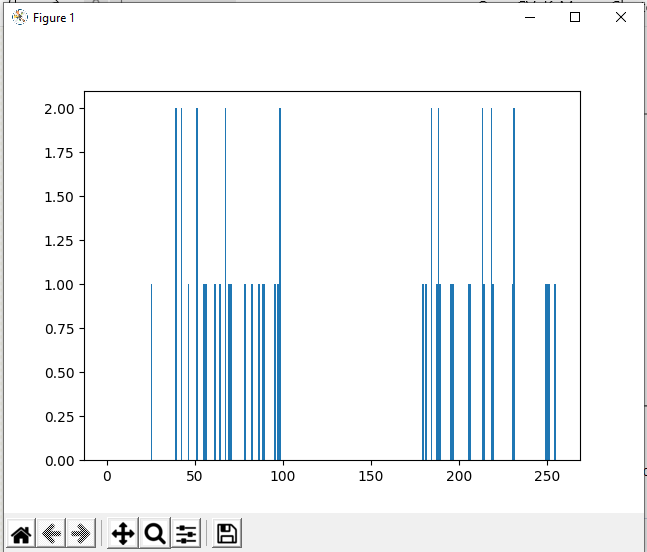
**y = np.random.randint(175,255,25)**

**z = np.hstack((x,y))**

**z = z.reshape((50,1))**

**z = np.float32(z)**

**plt.hist(z,256,[0,256]),plt.show()**



Теперь мы применяем функцию KMeans. Перед этим нам нужно уточнить критерии. Мои критерии таковы, что всякий раз, когда выполняется 10 итераций алгоритма или достигается точность Эпсилон = 1,0, остановите алгоритм и верните ответ.

**# Define criteria = ( type, max\_iter = 10 , epsilon = 1.0 )**

**criteria = (cv.TERM\_CRITERIA\_EPS + cv.TERM\_CRITERIA\_MAX\_ITER, 10, 1.0)**

**# Set flags (Just to avoid line break in the code)**

**flags = cv.KMEANS\_RANDOM\_CENTERS**

**# Apply KMeans**

**compactness,labels,centers = cv.kmeans(z,2,None,criteria,10,flags)**

Это дает нам компактность, метки и центры. В этом случае я получил центры как 60 и 207. Метки будут иметь тот же размер, что и тестовые данные, где каждая информация будет помечена как '0','1','2' и т.д. В зависимости от их центроидов. Теперь мы разделим данные на различные кластеры в зависимости от их меток.

**A = z[labels==0]**

**B = z[labels==1]**

Сейчас у нас участок в красный цвет и B в синий цвет и их центроиды в желтый цвет.

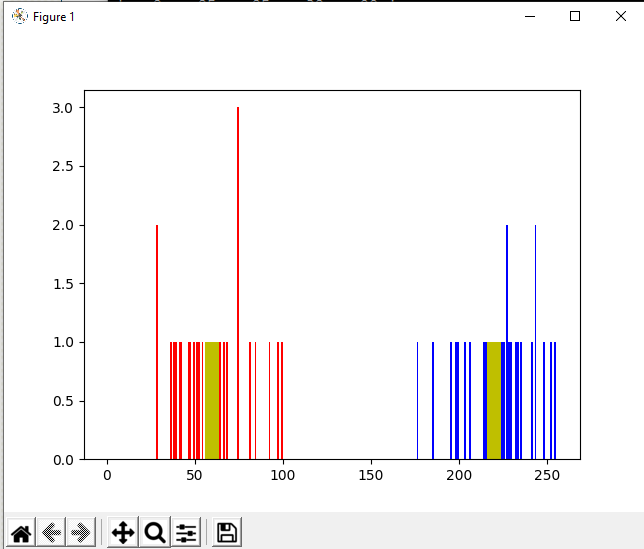
**# Now plot 'A' in red, 'B' in blue, 'centers' in yellow**

**plt.hist(A,256,[0,256],color = 'r')**

**plt.hist(B,256,[0,256],color = 'b')**

**plt.hist(centers,32,[0,256],color = 'y')**

**plt.show()**

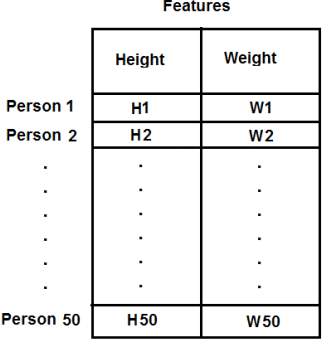


### 2. Данные с несколькими функциями

В предыдущем примере мы взяли только высоту для задачи о футболке. Здесь мы возьмем как рост, так и вес, то есть две особенности.

Помните, что в предыдущем случае мы сделали наши данные в один вектор столбца. Каждый признак расположен в столбце, а каждая строка соответствует входному тестовому образцу.

Например, в этом случае мы задаем тестовые данные размером 50х2, которые представляют собой рост и вес 50 человек. Первая колонка соответствует росту всех 50 человек, а вторая колонка соответствует их весу. Первый ряд содержит два элемента, где первый-это рост первого человека, а второй-его вес. Точно так же остальные ряды соответствуют росту и весу других людей. Проверьте изображение ниже:



Теперь я непосредственно перехожу к коду:

**import numpy as np**

**import cv2 as cv**

**from matplotlib import pyplot as plt**

**X = np.random.randint(25,50,(25,2))**

**Y = np.random.randint(60,85,(25,2))**

**Z = np.vstack((X,Y))**

**# convert to np.float32**

**Z = np.float32(Z)**

**# define criteria and apply kmeans()**

**criteria = (cv.TERM\_CRITERIA\_EPS + cv.TERM\_CRITERIA\_MAX\_ITER, 10, 1.0)**

**ret,label,center=cv.kmeans(Z,2,None,criteria,10,cv.KMEANS\_RANDOM\_CENTERS)**

**# Now separate the data, Note the flatten()**

**A = Z[label.ravel()==0]**

**B = Z[label.ravel()==1]**

**# Plot the data**

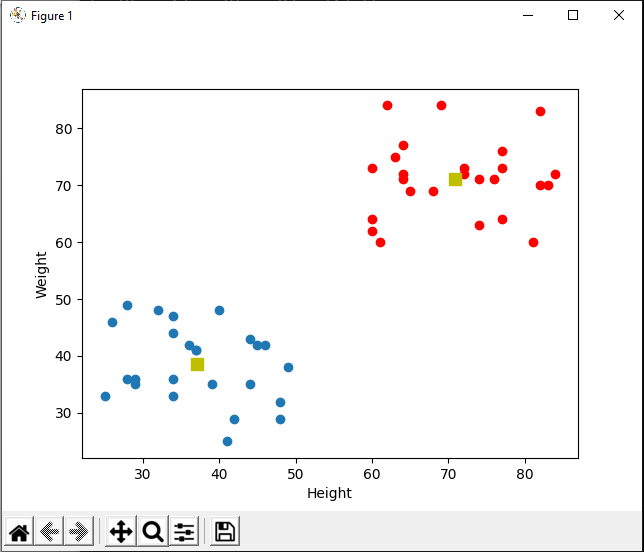
**plt.scatter(A[:,0],A[:,1])**

**plt.scatter(B[:,0],B[:,1],c = 'r')**

**plt.scatter(center[:,0],center[:,1],s = 80,c = 'y', marker = 's')**

**plt.xlabel('Height'),plt.ylabel('Weight')**

**plt.show()**



### 3. Цветовое Квантование

Квантование цвета - это процесс уменьшения количества цветов в изображении. Одна из причин этого-уменьшение объема памяти. Иногда некоторые устройства могут иметь такие ограничения, что они могут производить только ограниченное количество цветов. В этих случаях также выполняется квантование цвета. Здесь мы используем кластеризацию k-средних для квантования цвета.

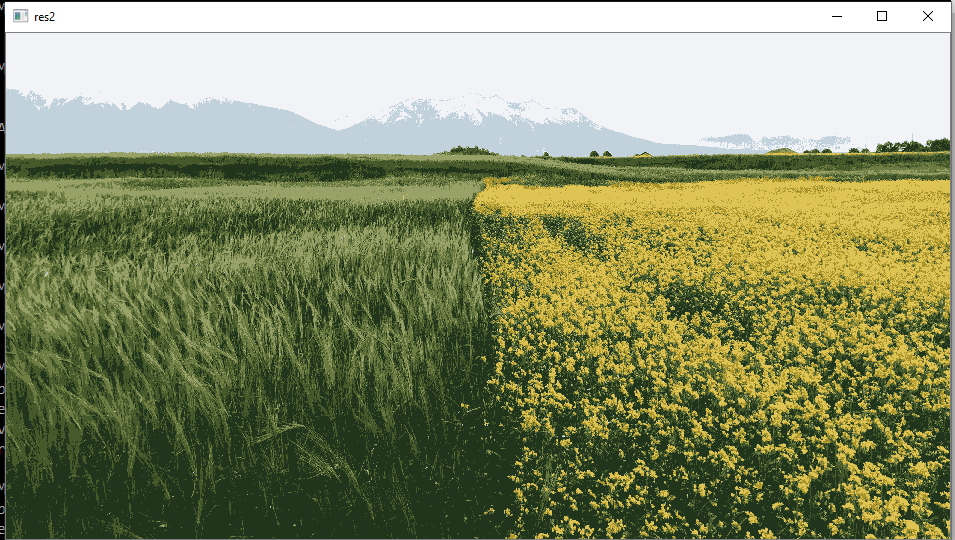
Здесь нет ничего нового, что можно было бы объяснить. Есть 3 функции, скажем, R,G,B. Поэтому нам нужно изменить форму изображения до массива размером Mx3 (M-количество пикселей в изображении). А после кластеризации мы применяем значения центроидов (это также R, G,B) ко всем пикселям,так что результирующее изображение будет иметь заданное количество цветов. И снова нам нужно изменить его обратно к форме исходного изображения. Ниже приведен код:



оригинал



4 цвета



8 цветов

Вопросы к лабораторной работе

1. привести подробное описание основных функций, рассмотренных в работе - функция, что делает, аргументы, параметры, возвращаемые результаты, типы данных и тп.
2. выполнить примеры, описанные в лабораторной - предоставить комментированный код, поэтапные результаты выполнения действий (исходные фото-, видео- данные для примеров берутся собственные, или из открытых источников или из базового набора библиотеки OpenCV)