**Overview**

**Document Scope**

This document is written to instruct, so it’s wordy.  The content is borrowed from Sentdex’s series on K-Means. The content is presented as notes about the code, generating visuals of plotted points and K-Means results. Enjoy the read and hopefully it provides some value.

Date: August 3, 2019

Author: John Askew

**Activity Prerequisites**

N/A

**Required Materials List**

**Environment Details**

Required Hardware

|  |  |  |
| --- | --- | --- |
| **SERVER** |  | **Purpose** |
| Linux or Windows |  | Host computer to run python |

Required Software

|  |  |  |
| --- | --- | --- |
| **SERVER/ENV** | **Software** | **Purpose** |
| Computer | Python 3, preferably 3.6+ | Exercise is written in Python 3 |
| Computer | matplotlib | Pyplot will provide visualizations for graph plotting |
| Computer | numpy | Python package to handle scientific oriented heavy lifting |
| Computer | os | Python package used to install missing Python packages |

## Code to install Python packages:

**import** os, sys

**try**:

**import** matplotlib.pyplot **as** plt

**except**:

os.system(**'pip install matplotlib**')

**import** matplotlib.pyplot **as** plt

**from** matplotlib **import** style

style.use(**'ggplot'**)

**try**:

**import** numpy **as** np

**except**:

os.system(**'pip install numpy'**)

**import** numpy **as** np

## Review Coding Logic

#### PROCEDURE 0: Review what we are doing with K-Means

|  |  |  |
| --- | --- | --- |
| **Step** | 🗹 | **Action / Expected Result** |
|  |  | **Action**: Review purpose of K-Means and point of this exercise   * 1. K-Means. *Unsupervised Machine Learning, using clustering to find groups within the data.*   [*https://www.datascience.com/blog/k-means-clustering*](https://www.datascience.com/blog/k-means-clustering)   * 1. What is the point of this ? *Introduce machine learning using a K-Means engine we code ourselves, to better understand the innerworkings of the K-Means algorithm..*   2. What is the end result?. *We end up with a visual of our data being grouped via the K-Means logic.* |
|  |  | End of section |

#### PROCEDURE 1: Install software for Python to run this exercise

| **Step** | 🗹 | **Action / Expected Result** |
| --- | --- | --- |
|  |  | **Action**: Review code installing Python software.  **Commands (code):**  **import** os, sys  **try**:  **import** matplotlib.pyplot **as** plt  **except**:  os.system(**'pip install matplotlib**')  **import** matplotlib.pyplot **as** plt  **from** matplotlib **import** style  style.use(**'ggplot'**)  **try**:  **import** numpy **as** np  **except**:  os.system(**'pip install numpy'**)  **import** numpy **as** np  **Expected Results:**  Any missing packages needed to run this code will be installed and instantiated. See the *Required Materials List* section at the beginning of this document for additional details. |
| 2. |  | End of section |

#### PROCEDURE 1: Review data and data format

| **Step** | 🗹 | **Action / Expected Result** |
| --- | --- | --- |
| 1. |  | **Action**: Understand the data and the array format  *Each element is simple a point on the Cartesian plane (a simple graph with x & y axis).*  **Code:**  X = np.array( [ [1, 2],  [1.5, 1.8],  [5, 8],  [8, 8],  [1, .6],  [9, 11],  [8, 2],  [10, 2],  [9, 3],  [11,4],  [2, 1]]  *The first element is the point: 1, 2 – where x=1 and y = 2. The second element is point 1.5, 1.8, where x=1.5 and y = 1.8.* |
| 2. |  | **Action**: Understand the array format  *Python currently does not have a default or generic array format. We use the numpy package to generate an array. The array is generally more efficient than Python lists – based on space consumed and processing speed.*  **Code:**  X = **np.array**( [ [1, 2],  [1.5, 1.8],  [5, 8],  [8, 8],  [1, .6],  [9, 11],  [8, 2],  [10, 2],  [9, 3],  [11,4],  [2, 1]] |

#### PROCEDURE 2: Review the code

| **Step** | 🗹 | **Action / Expected Result** |
| --- | --- | --- |
| 1. |  | **Action**: Review the first pass of the code  The array named X (which contains our data) is loaded into a dictionary named points.  The first point (1, 2) is compared to itself (the first data item or element in the dictionary named points.). and naturally generates a 0 length distance. The distance between anything and itself is always 0.  The code loops thru the remaining data items in points and compares it to the first data item, (1, 2).  **Example:**  (1, 2) is the first point.  (1.5, 1.8) is the next data item in points.  We compare the distance between points (1, 2) and (1.5, 1.8). The Euclidean distance is the distance between (1, 2) and (1.5, 1.8). The result is 0.5385 – meaning the length of the imaginary line joining (1, 2) and (1.5, 1.8).  Each data item in points is compared to the point (1,2) and the Euclidean distance is computed, meaning the distance between (1, 2) and each data item in points. Each Euclidean distance is then evaluated to see if it falls within a certain range (less than 4). Since the distance between (1, 2) and (1.5, 1.8) is .**5385** and is less than 4, we save off the (1.5, 1.8) in a holding area named, **this\_group**.  This process of computing and storing the data items whose Euclidean distance is less than 4 continues until every data item in X is processed. When completed, we have multiple saved off data points, with duplicates, so we remove the duplicates. Once our saved off data items contain unique points, we take the average (mean) of the data points and this is saved off as the first potential center point, named new\_centroid.  Given the first code execution pass for point (1, 2), we have saved off data points: (1,2 ), (1.5, 1.8), (1, .6), (2, 1). The average (Euclidean distance) of all these datapoints is (1.375, 1.35).  The above code is repeated for all elements (points) in the dictionary named points. When we have processed all the data, the saved off Euclidean distances are:  [(1.375, 1.3499999999999999), (1.375, 1.3499999999999999), (6.5, 8.0), (7.333333333333333, 9.0), (1.375, 1.3499999999999999), (8.5, 9.5), (9.5, 2.75), (9.5, 2.75), (9.5, 2.75), (9.5, 2.75), (1.375, 1.3499999999999999)].  This, of course, needs to have duplicates removed before verifying whether each saved off Euclidean distance truly represents the center point of each group. Ending result for the first execution of the code is: (1.375, 1.3499999999999999), (6.5, 8.0), (7.333333333333333, 9.0), (8.5, 9.5), (9.5, 2.75)]  This leads us to the NEXT execution (pass through) of the code. |
| 2. |  | **Action**: Review repeated execution (loop) of the code.  The saved off Euclidean distances are loaded into the dictionary named points and the comparison between each item in points is compared to each element in the array named data (which is originally defined as array, “X”). New Euclidean distances are computed and optimized.  **Second iteration example:** Dictionary points is loaded with: **{0: array([**1.375, 1.35 **]), 1: array([**6.5, 8. **]), 2: array([**7.33333333, 9. **]), 3: array([**8.5, 9.5**]), 4: array([**9.5 , 2.75**])}**  **Third iteration example:**  Dictionary points are loaded with: **{0: array([**1.375, 1.35 **]), 1: array([**7.33333333, 9. **]), 2: array([**9.5 , 2.75**])}**  **Expected Results:**  **The Euclidean centers are optimized for each group of points and additional passes through the code will not change the calculated Euclidean distances.**  **Two visuals are produced: 1. Data points are plotted on a graph; 2. The K-Means results is displayed on the graph showing groups and the centers of each group. We have now taken raw data through an unsupervised model and grouped similar data points. Each group technically would represent a data value contained in a column in a table, also known as an attribute or feature** |

#### PROCEDURE 3: Present code

import os, sys

try:

import matplotlib.pyplot as plt

except:

os.system('pip install matplotlib')

import matplotlib.pyplot as plt

from matplotlib import style

style.use('ggplot')

try:

import numpy as np

except:

os.system('pip install numpy')

import numpy as np

X = np.array( [ [1, 2],

[1.5, 1.8],

[5, 8],

[8, 8],

[1, .6],

[9, 11],

[8, 2],

[10, 2],

[9, 3],

[11,4],

[2, 1]

]

)

# colors =int((len(X)/3) + int(len(X)%3)) \* ('g', 'r', 'm','bisque', 'forestgreen', 'slategrey' )

color\_list = []

colors = ('g', 'r', 'olivedrab', 'salmon', 'hotpink' , 'thistle',

'dimgray', 'darkorange', 'turquoise' , 'lightsteelblue',

'burlywood', 'royalblue','midnightblue', 'darkgoldenrod',

'lightseagreen', 'mediumslateblue', 'rosybrown' , 'limegreen',

'coral', 'yellow', 'powderblue', 'fuschia',

'orangered', 'orchid', 'deepskyblue','forestgreen')

for cnt, color in enumerate(colors):

if cnt < int(len(X)):

color\_list.append(colors[cnt])

class Mean\_Shift:

def \_\_init\_\_(self, radius = 4):

self.radius = radius

def fit(self, data):

points = {} # using the dict command to set points[i] = data[i] and then prev\_points = dict(points)

for i in range(len(data)):

points[i] = data[i]

while True:

new\_centroids = []

for i in points:

this\_group = []

point = points[i]

for featureset in data: # featureset is the index to X. First entry is featureset: [1 2]. Data is whole array

if np.linalg.norm(featureset - point) < self.radius:

this\_group.append(featureset) # Save any point with eud. dist. < radius (set to 4 by default)

new\_centroid = np.average(this\_group, axis = 0) # Avg. of all stored points in this\_group

new\_centroids.append(tuple(new\_centroid))

uniques = sorted(list(set(new\_centroids)))

prev\_points = dict(points)

points = {}

for i in range(len(uniques)):

points[i] = np.array(uniques[i])

optimized = True

for i in points:

if not np.array\_equal(points[i], prev\_points[i]):

optimized = False

if not optimized:

break

if optimized:

break

self.points = points

def predict(self, data):

pass

#######################################

# MAIN LOGIC

#######################################

clf = Mean\_Shift() #Instantiate our model or classifer as the Mean\_Shift

clf.fit(X) #Call the fit code, listed above, which constitutes the majority of the code.

points = clf.points # Define points (outside the class) as pointing to the class' self.points = points

plt.scatter(X[:, 0], X[:,1], s = 100, color = color\_list)

plt.title("Raw Data before calculating KMeans")

plt.show()

plt.scatter(X[:, 0], X[:,1], s = 100, color = color\_list)

for c in points:

plt.scatter(points[c][0], points[c][1], color = 'k', marker = '\*', s=250)

plt.annotate('center', (points[c][0]+.10, points[c][1] +.15), size=8)

plt.annotate((round(points[c][0],2),round(points[c][1],2)), (points[c][0]+ .15, points[c][1] -.18), size=8)

circlex = plt.Circle((points[c][0] , points[c][1]), 2.9, color = 'gold', alpha = .1)

plt.gcf().gca().add\_artist(circlex) # gcf = get current figure; gca = get current axes (ax). Replaces fig, ax = plt.figure, ax = ???

plt.title("KMeans clustering into groups")

plt.show()

END OF DOCUMENT