Sample solutions

Clustering via k-means

We previously studied the classification problem using the logistic regression algorithm. Since we had labels for each data point, we may regard as one of *supervised learning*. However, in many applications, the data have no labels but we wish to discover possible labels (or other hidden p structures). This problem is one of *unsupervised learning*. How can we approach such problems?

Clustering is one class of unsupervised learning methods. In this lab, we'll consider the following form of the clustering task. Suppose you are g

- a set of observations, $X \equiv \{\hat{x}_i \,|\, 0 \leq i < n\}$, and
- a target number of *clusters*, k.

Your goal is to partition the points into k subsets, $C_0, \ldots, C_{k-1} \subseteq X$, which are

- disjoint, i.e., $i \neq j \implies C_i \cap C_j = \emptyset$;
- but also complete, i.e., $C_0 \cup C_1 \cup \cdots \cup C_{k-1} = X$.

Intuitively, each cluster should reflect some "sensible" grouping. Thus, we need to specify what constitutes such a grouping.

Setup: Dataset

The following cell will download the data you'll need for this lab. Run it now.

In [1]:
```
import requests
import os
import hashlib
import io

def on_vocareum():
    return os.path.exists('.voc')

def download(file, local_dir="", url_base=None, checksum=None):
    local_file = "{}{}".format(local_dir, file)
    if not os.path.exists(local_file):
        if url_base is None:
            url_base = "https://cse6040.gatech.edu/datasets/"
        url = "{}{}".format(url_base, file)
        print("Downloading: {} ...".format(url))
        r = requests.get(url)
        with open(local_file, 'wb') as f:
            f.write(r.content)

    if checksum is not None:
        with io.open(local_file, 'rb') as f:
            body = f.read()
            body_checksum = hashlib.md5(body).hexdigest()
            assert body_checksum == checksum, \
                "Downloaded file '{}' has incorrect checksum: '{}' instead of '{}'".format(local_file,
                                                                                             body_checksu
                                                                                             checksum)
    print("'{}' is ready!".format(file))

if on_vocareum():
    URL_BASE = "https://cse6040.gatech.edu/datasets/kmeans/"
    DATA_PATH = "../resource/asnlib/publicdata/"
else:
    URL_BASE = "https://github.com/cse6040/labs-fa17/raw/master/datasets/kmeans/"
    DATA_PATH = ""

datasets = {'logreg_points_train.csv': '9d1e42f49a719da43113678732491c6d',
            'centers_initial_testing.npy': '8884b4af540c1d5119e6e8980da43f04',
            'compute_d2_soln.npy': '980fe348b6cba23cb81ddf703494fb4c',
            'y_test3.npy': 'df322037ea9c523564a5018ea0a70fbf',
            'centers_test3_soln.npy': '0c594b28e512a532a2ef4201535868b5',
            'assign_cluster_labels_S.npy': '37e464f2b79dc1d59f5ec31eaefe4161',
            'assign_cluster_labels_soln.npy': 'fc0e084ac000f30948946d097ed85ebc'}

for filename, checksum in datasets.items():
    download(filename, local_dir=DATA_PATH, url_base=URL_BASE, checksum=checksum)

print("\n(All data appears to be ready.)")
```

```
'logreg_points_train.csv' is ready!
'y_test3.npy' is ready!
'compute_d2_soln.npy' is ready!
'assign_cluster_labels_soln.npy' is ready!
'centers_test3_soln.npy' is ready!
'assign_cluster_labels_S.npy' is ready!
'centers_initial_testing.npy' is ready!
```

```
'centers_initial_testing.npy' is ready!

(All data appears to be ready.)
```

The k-means clustering criterion

Here is one way to measure the quality of a set of clusters. For each cluster C, consider its center μ and measure the distance $\|x - \mu\|$ of eac $x \in C$ to the center. Add these up for all points in the cluster; call this sum is the *within-cluster sum-of-squares (WCSS)*. Then, set as our goal t clusters that minimize the total WCSS over *all* clusters.

More formally, given a clustering $C = \{C_0, C_1, \ldots, C_{k-1}\}$, let

$$\mathrm{WCSS}(C) \equiv \sum_{i=0}^{k-1} \sum_{x \in C_i} \|x - \mu_i\|^2,$$

where μ_i is the center of C_i. This center may be computed simply as the mean of all points in C_i, i.e.,

$$\mu_i \equiv \frac{1}{|C_i|} \sum_{x \in C_i} x.$$

Then, our objective is to find the "best" clustering, C_*, which is the one that has a minimum WCSS.

$$C_* = \arg\min_C \mathrm{WCSS}(C).$$

The standard k-means algorithm (Lloyd's algorithm)

Finding the global optimum is NP-hard (https://en.wikipedia.org/wiki/NP-hardness), which is computer science mumbo jumbo for "we don't know is an algorithm to calculate the exact answer in fewer steps than exponential in the size of the input." Nevertheless, there is an iterative method, algorithm, that can quickly converge to a *local* (as opposed to *global*) minimum. The procedure alternates between two operations: *assignment a*

Step 1: Assignment. Given a fixed set of k centers, assign each point to the nearest center:

$$C_i = \{\hat{x} : \|\hat{x} - \mu_i\| \leq \|\hat{x} - \mu_j\|, 1 \leq j \leq k\}.$$

Step 2: Update. Recompute the k centers ("centroids") by averaging all the data points belonging to each cluster, i.e., taking their mean:

$$\mu_i = \frac{1}{|C_i|} \sum_{\hat{x} \in C_i} \hat{x}$$

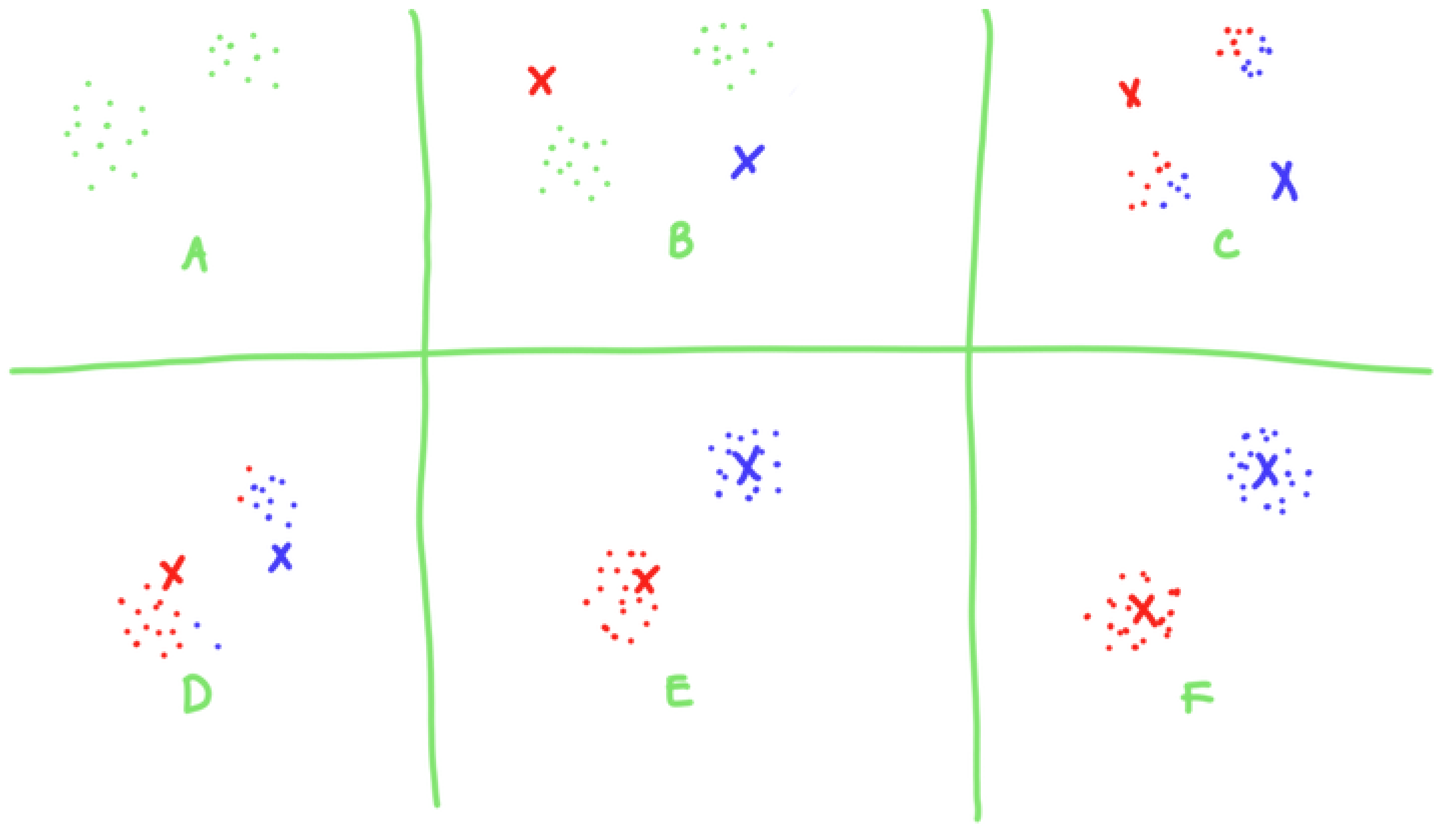


Figure adapted from: http://stanford.edu/~cpiech/cs221/img/kmeansViz.png (http://stanford.edu/~cpiech/cs221/img/kmeansViz.png)

In the code that follows, it will be convenient to use our usual "data matrix" convention, that is, each row of a data matrix X is one of m observa column (coordinate) is one of d predictors. However, we will *not* need a dummy column of ones since we are not fitting a function.

$$X \equiv \begin{pmatrix} \hat{x}_0^T \\ \vdots \\ \hat{x}_m^T \end{pmatrix} = \begin{pmatrix} x_0 & \cdots & x_{d-1} \end{pmatrix}.$$

In [2]:
```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

%matplotlib inline
```

```
%matplotlib inline

import matplotlib as mpl
mpl.rc("savefig", dpi=100) # Adjust for higher-resolution figures
```

We will use the following data set which some of you may have seen previously.

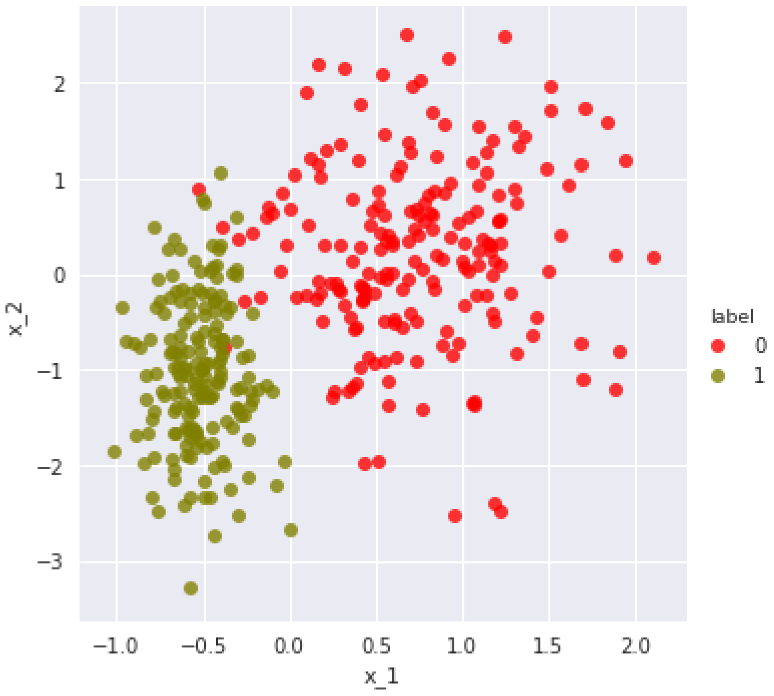
In [3]:
```
df = pd.read_csv('{}logreg_points_train.csv'.format(DATA_PATH))
df.head()
```

Out[3]:

	x_1	x_2	label
0	-0.234443	-1.075960	1
1	0.730359	-0.918093	0
2	1.432270	-0.439449	0
3	0.026733	1.050300	0
4	1.879650	0.207743	0

In [4]:
```
# Helper functions from Logistic Regression Lesson
def make_scatter_plot(df, x="x_1", y="x_2", hue="label",
                      palette={0: "red", 1: "olive"},
                      size=5,
                      centers=None):
    sns.lmplot(x=x, y=y, hue=hue, data=df, palette=palette,
               fit_reg=False)
    if centers is not None:
        plt.scatter(centers[:,0], centers[:,1],
                    marker=u'*', s=500,
                    c=[palette[0], palette[1]])

def mark_matches(a, b, exact=False):
    """
    Given two Numpy arrays of {0, 1} labels, returns a new boolean
    array indicating at which locations the input arrays have the
    same label (i.e., the corresponding entry is True).

    This function can consider "inexact" matches. That is, if `exact`
    is False, then the function will assume the {0, 1} labels may be
    regarded as the same up to a swapping of the labels. This feature
    allows

      a == [0, 0, 1, 1, 0, 1, 1]
      b == [1, 1, 0, 0, 1, 0, 0]

    to be regarded as equal. (That is, use `exact=False` when you
    only care about "relative" labeling.)
    """
    assert a.shape == b.shape
    a_int = a.astype(dtype=int)
    b_int = b.astype(dtype=int)
    all_axes = tuple(range(len(a.shape)))
    assert ((a_int == 0) | (a_int == 1)).all()
    assert ((b_int == 0) | (b_int == 1)).all()

    exact_matches = (a_int == b_int)
    if exact:
        return exact_matches

    assert exact == False
    num_exact_matches = np.sum(exact_matches)
    if (2*num_exact_matches) >= np.prod (a.shape):
        return exact_matches
    return exact_matches == False # Invert

def count_matches(a, b, exact=False):
    """
    Given two sets of {0, 1} labels, returns the number of mismatches.

    This function can consider "inexact" matches. That is, if `exact`
    is False, then the function will assume the {0, 1} labels may be
    regarded as similar up to a swapping of the labels. This feature
    allows

      a == [0, 0, 1, 1, 0, 1, 1]
      b == [1, 1, 0, 0, 1, 0, 0]

    to be regarded as equal. (That is, use `exact=False` when you
    only care about "relative" labeling.)
    """
    matches = mark_matches(a, b, exact=exact)
    return np.sum(matches)
```

In [5]:
```
make_scatter_plot(df)
```



Let's extract the data points as a data matrix, `points`, and the labels as a vector, `labels`. Note that the k-means algorithm you will implement sh
reference `labels` -- that's the solution we will try to predict given only the point coordinates (`points`) and target number of clusters (`k`).

```
In [6]: points = df.as_matrix(['x_1', 'x_2'])
        labels = df['label'].as_matrix()
        n, d = points.shape
        k = 2
```

Note that the labels should *not* be used in the k-means algorithm. We use them here only as ground truth for later verification.

How to start? Initializing the k centers

To start the algorithm, you need an initial guess. Let's randomly choose k observations from the data.

Exercise 1 (2 points). Complete the following function, `init_centers(X, k)`, so that it randomly selects k of the given observations to serve a
should return a Numpy array of size `k`-by-`d`, where `d` is the number of columns of `X`.

```
In [7]: def init_centers(X, k):
            """
            Randomly samples k observations from X as centers.
            Returns these centers as a (k x d) numpy array.
            """
            ### BEGIN SOLUTION
            from numpy.random import choice
            samples = choice(len(X), size=k, replace=False)
            return X[samples, :]
            ### END SOLUTION
```

```
In [8]: # Test cell: `init_centers_test`

        centers_initial = init_centers(points, k)
        print("Initial centers:\n", centers_initial)

        assert type(centers_initial) is np.ndarray, "Your function should return a Numpy array instead of a {}"
        (centers_initial))
        assert centers_initial.shape == (k, d), "Returned centers do not have the right shape ({} x {})".format
        assert (sum(centers_initial[0, :] == points) == [1, 1]).all(), "The centers must come from the input."
        assert (sum(centers_initial[1, :] == points) == [1, 1]).all(), "The centers must come from the input."

        print("\n(Passed!)")
```

```
Initial centers:
 [[ 0.428191 -1.9734   ]
 [ 0.75525   2.03587 ]]

(Passed!)
```

Computing the distances

Exercise 2 (3 points). Implement a function that computes a distance matrix, $S = (s_{ij})$ such that $s_{ij} = d_{ij}^2$ is the *squared* distance from point
μ_j. It should return a Numpy matrix `S[:m, :k]`.

```
In [9]: def compute_d2(X, centers):
            m = len(X)
            k = len(centers)
```

```
    S = np.empty((m, k))

    ### BEGIN SOLUTION
    for i in range(m):
        d_i = np.linalg.norm(X[i, :] - centers, ord=2, axis=1)
        S[i, :] = d_i**2
    ### END SOLUTION

    return S
```

In [10]:
```
# Test cell: `compute_d2_test`

centers_initial_testing = np.load("{}centers_initial_testing.npy".format(DATA_PATH))
compute_d2_soln = np.load("{}compute_d2_soln.npy".format(DATA_PATH))

S = compute_d2 (points, centers_initial_testing)
assert (np.linalg.norm (S - compute_d2_soln, axis=1) <= (20.0 * np.finfo(float).eps)).all ()

print("\n(Passed!)")
```

```
(Passed!)
```

Exercise 3 (2 points). Write a function that uses the (squared) distance matrix to assign a "cluster label" to each point.

That is, consider the $m \times k$ squared distance matrix S. For each point i, if $s_{i,j}$ is the minimum squared distance for point i, then the index j is label. In other words, your function should return a (column) vector y of length m such that

$$y_i = \underset{j \in \{0,\ldots,k-1\}}{\operatorname{argmin}} \; s_{ij}.$$

> Hint: Judicious use of Numpy's `argmin()` (https://docs.scipy.org/doc/numpy/reference/generated/numpy.argmin.html) makes for a nice c solution.

In [11]:
```
def assign_cluster_labels(S):
    ### BEGIN SOLUTION
    return np.argmin(S, axis=1)
    ### END SOLUTION

# Cluster labels:     0    1
S_test1 = np.array([[0.3, 0.2],  # --> cluster 1
                    [0.1, 0.5],  # --> cluster 0
                    [0.4, 0.2]]) # --> cluster 1
y_test1 = assign_cluster_labels(S_test1)
print("You found:", y_test1)

assert (y_test1 == np.array([1, 0, 1])).all()
```

```
You found: [1 0 1]
```

In [12]:
```
# Test cell: `assign_cluster_labels_test`

S_test2 = np.load("{}assign_cluster_labels_S.npy".format(DATA_PATH))
y_test2_soln = np.load("{}assign_cluster_labels_soln.npy".format(DATA_PATH))
y_test2 = assign_cluster_labels(S_test2)
assert (y_test2 == y_test2_soln).all()

print("\n(Passed!)")
```

```
(Passed!)
```

Exercise 4 (2 points). Given a clustering (i.e., a set of points and assignment of labels), compute the center of each cluster.

In [13]:
```
def update_centers(X, y):
    # X[:m, :d] == m points, each of dimension d
    # y[:m] == cluster labels
    m, d = X.shape
    k = max(y) + 1
    assert m == len(y)
    assert (min(y) >= 0)

    centers = np.empty((k, d))
    for j in range(k):
        # Compute the new center of cluster j,
        # i.e., centers[j, :d].
        ### BEGIN SOLUTION
        centers[j, :d] = np.mean(X[y == j, :], axis=0)
        ### END SOLUTION
    return centers
```

In [14]:
```
# Test cell: `update_centers_test`

y_test3 = np.load("{}y_test3.npy".format(DATA_PATH))
centers_test3_soln = np.load("{}centers_test3_soln.npy".format(DATA_PATH))
```

```
centers_test3 = update_centers(points, y_test3)

delta_test3 = np.abs(centers_test3 - centers_test3_soln)
assert (delta_test3 <= 2.0*len(centers_test3_soln)*np.finfo(float).eps).all()

print("\n(Passed!)")
```

```
(Passed!)
```

Exercise 5 (2 points). Given the squared distances, return the within-cluster sum of squares.

In particular, your function should have the signature,

```
def WCSS(S):
    ...
```

where `S` is an array of distances as might be computed from Exercise 2.

For example, suppose `S` is defined as follows:

```
S = np.array([[0.3, 0.2],
              [0.1, 0.5],
              [0.4, 0.2]])
```

Then `WCSS(S) == 0.2 + 0.1 + 0.2 == 0.5`.

> *Hint*: See numpy.amin (https://docs.scipy.org/doc/numpy/reference/generated/numpy.amin.html#numpy.amin).

In [15]:
```
def WCSS(S):
    ### BEGIN SOLUTION
    return np.sum(np.amin(S, axis=1))
    ### END SOLUTION

# Quick test:
print("S ==\n", S_test1)
WCSS_test1 = WCSS(S_test1)
print("\nWCSS(S) ==", WCSS(S_test1))
```

```
S ==
 [[0.3 0.2]
 [0.1 0.5]
 [0.4 0.2]]

WCSS(S) == 0.5
```

In [16]:
```
# Test cell: `WCSS_test`

assert np.abs(WCSS_test1 - 0.5) <= 3.0*np.finfo(float).eps, "WCSS(S_test1) should be close to 0.5, not
CSS_test1)
print("\n(Passed!)")
```

```
(Passed!)
```

Lastly, here is a function to check whether the centers have "moved," given two instances of the center values. It accounts for the fact that the or
may have changed.

In [17]:
```
def has_converged(old_centers, centers):
    return set([tuple(x) for x in old_centers]) == set([tuple(x) for x in centers])
```

Exercise 6 (3 points). Put all of the preceding building blocks together to implement Lloyd's k-means algorithm.

In [18]:
```
def kmeans(X, k,
           starting_centers=None,
           max_steps=np.inf):
    if starting_centers is None:
        centers = init_centers(X, k)
    else:
        centers = starting_centers

    converged = False
    labels = np.zeros(len(X))
    i = 1
    while (not converged) and (i <= max_steps):
        old_centers = centers
        ### BEGIN SOLUTION
        S = compute_d2(X, centers)
        labels = assign_cluster_labels(S)
        centers = update_centers(X, labels)
        converged = has_converged(old_centers, centers)
        ### END SOLUTION
        print ("iteration", i, "WCSS = ", WCSS (S))
```

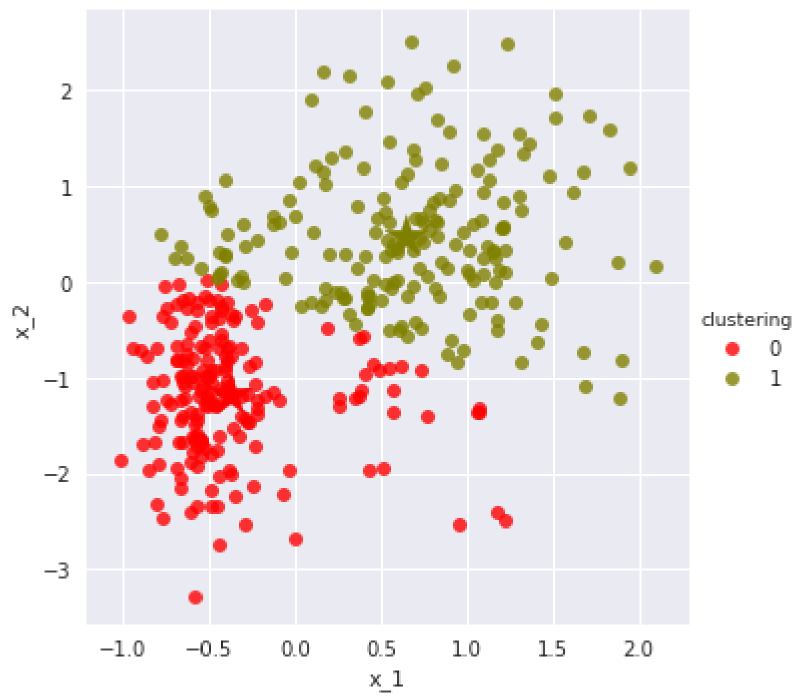
```
        i += 1
    return labels

clustering = kmeans(points, k, starting_centers=points[[0, 187], :])
```

```
iteration 1 WCSS =  549.9175535488309
iteration 2 WCSS =  339.80066330255096
iteration 3 WCSS =  300.330112922328
iteration 4 WCSS =  289.80700777322045
iteration 5 WCSS =  286.0745591062787
iteration 6 WCSS =  284.1907705579879
iteration 7 WCSS =  283.22732249939105
iteration 8 WCSS =  282.456491302569
iteration 9 WCSS =  281.84838225337074
iteration 10 WCSS =  281.57242082723724
iteration 11 WCSS =  281.5315627987326
```

Let's visualize the results.

In [19]:
```
# Test cell: `kmeans_test`

df['clustering'] = clustering
centers = update_centers(points, clustering)
make_scatter_plot(df, hue='clustering', centers=centers)

n_matches = count_matches(df['label'], df['clustering'])
print(n_matches,
      "matches out of",
      len(df), "possible",
      "(~ {:.1f}%)".format(100.0 * n_matches / len(df)))

assert n_matches >= 320
```

```
329 matches out of 375 possible (~ 87.7%)
```



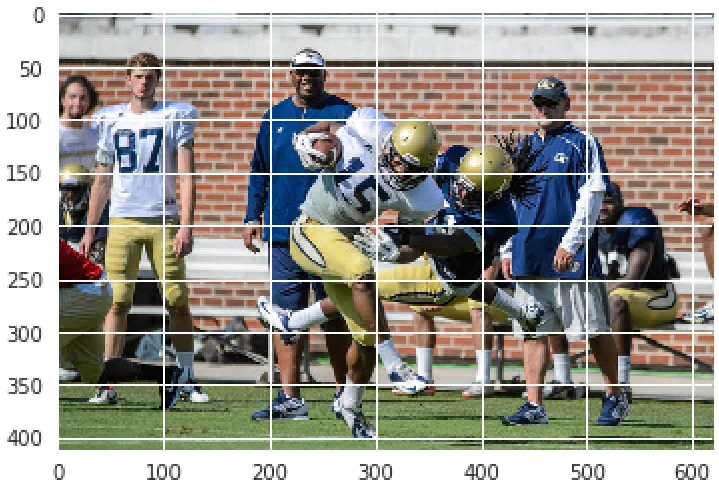
Applying k-means to an image. In this section of the notebook, you will apply k-means to an image, for the purpose of doing a "stylized recolor can view this example as a primitive form of [artistic style transfer (http://genekogan.com/works/style-transfer/)](http://genekogan.com/works/style-transfer/), which state-of-the-art methods to accomplish using neural networks (https://medium.com/artists-and-machine-intelligence/neural-artistic-style-transfer-a-comprehensive-look-f54d

In particlar, let's take an input image and cluster pixels based on the similarity of their colors. Maybe it can become the basis of your own Instagr (https://blog.hubspot.com/marketing/instagram-filters)!

In [20]:
```
from PIL import Image
from matplotlib.pyplot import imshow
%matplotlib inline

def read_img(path):
    """
    Read image and store it as an array, given the image path.
    Returns the 3 dimensional image array.
    """
    img = Image.open(path)
    img_arr = np.array(img, dtype='int32')
    img.close()
    return img_arr

def display_image(arr):
    """
    display the image
    input : 3 dimensional array
    """
    arr = arr.astype(dtype='uint8')
    img = Image.fromarray(arr, 'RGB')
    imshow(np.asarray(img))
```

```
img_arr = read_img("../resource/asnlib/publicdata/football.bmp")
display_image(img_arr)
print("Shape of the matrix obtained by reading the image")
print(img_arr.shape)
```

```
Shape of the matrix obtained by reading the image
(412, 620, 3)
```



Note that the image is stored as a "3-D" matrix. It is important to understand how matrices help to store a image. Each pixel corresponds to a int Red, Green and Blue. If you note the properties of the image, its resolution is 620 x 412. The image width is 620 pixels and height is 412 pixels, has three values - **R**, **G**, **B**. This makes it a 412 x 620 x 3 matrix.

Exercise 7 (1 point). Write some code to *reshape* the matrix into "img_reshaped" by transforming "img_arr" from a "3-D" matrix to a flattened "2- which has 3 columns corresponding to the RGB values for each pixel. In this form, the flattened matrix must contain all pixels and their correspo intensity values. Remember in the previous modules we had discussed a C type indexing style and a Fortran type indexing style. In this problem type indexing style. The numpy reshape function may be of help here.

In [21]:
```
### BEGIN SOLUTION
r, c, l = img_arr.shape
img_reshaped = np.reshape(img_arr, (r*c, l), order="C")
### END SOLUTION
```

In [22]:
```
# Test cell - 'reshape_test'
r, c, l = img_arr.shape
# The reshaped image is a flattened '2-dimensional' matrix
assert len(img_reshaped.shape) == 2
r_reshaped, c_reshaped = img_reshaped.shape
assert r * c * l == r_reshaped * c_reshaped
assert c_reshaped == 3
print("Passed")
```

```
Passed
```

Exercise 8 (1 point). Now use the k-means function that you wrote above to divide the image in **3** clusters. The result would be a vector which a label to each pixel.

In [23]:
```
### BEGIN SOLUTION
labels = kmeans(img_reshaped, 3)
### END SOLUTION
```

```
iteration 1 WCSS =  3191006513.0
iteration 2 WCSS =  887886047.4271191
iteration 3 WCSS =  669086576.3837116
iteration 4 WCSS =  640418622.8330001
iteration 5 WCSS =  636366884.6415913
iteration 6 WCSS =  635141015.9468135
iteration 7 WCSS =  634601099.9963626
iteration 8 WCSS =  634372413.2726401
iteration 9 WCSS =  634266793.5137541
iteration 10 WCSS =  634214992.2303745
iteration 11 WCSS =  634190480.2853614
iteration 12 WCSS =  634179610.7516326
iteration 13 WCSS =  634176205.8989807
iteration 14 WCSS =  634174590.2671936
iteration 15 WCSS =  634173984.4367541
iteration 16 WCSS =  634173813.1828784
iteration 17 WCSS =  634173778.4150583
iteration 18 WCSS =  634173759.7959646
iteration 19 WCSS =  634173756.0509284
iteration 20 WCSS =  634173755.1569637
```

In [24]:
```
# Test cell - 'labels'
assert len(labels) == r_reshaped
assert set(labels) == {0, 1, 2}
print("\nPassed!")
```

```
Passed!
```

Exercise 9 (2 points). Write code to calculate the mean of each cluster and store it in a dictionary as label:array(cluster_center). For 3 clusters, t should have three keys as the labels and their corresponding cluster centers as values, i.e. {0:array(center0), 1: array(center1), 2:array(center2)

In [25]:
```
### BEGIN SOLUTION
ind = np.column_stack((img_reshaped, labels))
centers = {}
for i in set(labels):
    c = ind[ind[:,3] == i].mean(axis=0)
    centers[i] = c[:3]
### END SOLUTION
```

In [26]:
```
print("Free points here! But you need to implement the above section correctly for you to see what we w
ee later.")
print("\nPassed!")
```

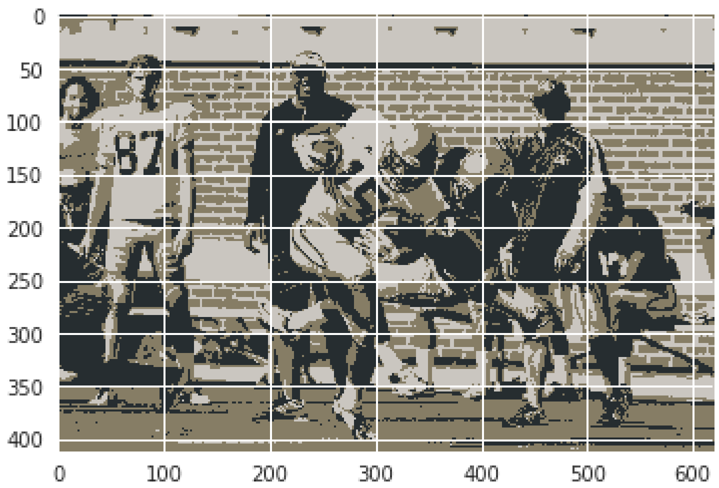
```
Free points here! But you need to implement the above section correctly for you to see what we want you
er.

Passed!
```

Below, we have written code to generate a matrix "img_clustered" of the same dimensions as img_reshaped, where each pixel is replaced by th center to which it belongs.

In [27]:
```
img_clustered = np.array([centers[i] for i in labels])
```

Let us display the clustered image and see how kmeans works on the image.

In [28]:
```
r, c, l = img_arr.shape
img_disp = np.reshape(img_clustered, (r, c, l), order="C")
display_image(img_disp)
```



You can visually inspect the original image and the clustered image to get a sense of what kmeans is doing here. You can also try to vary the nu clusters to see how the output image changes

Built-in k-means

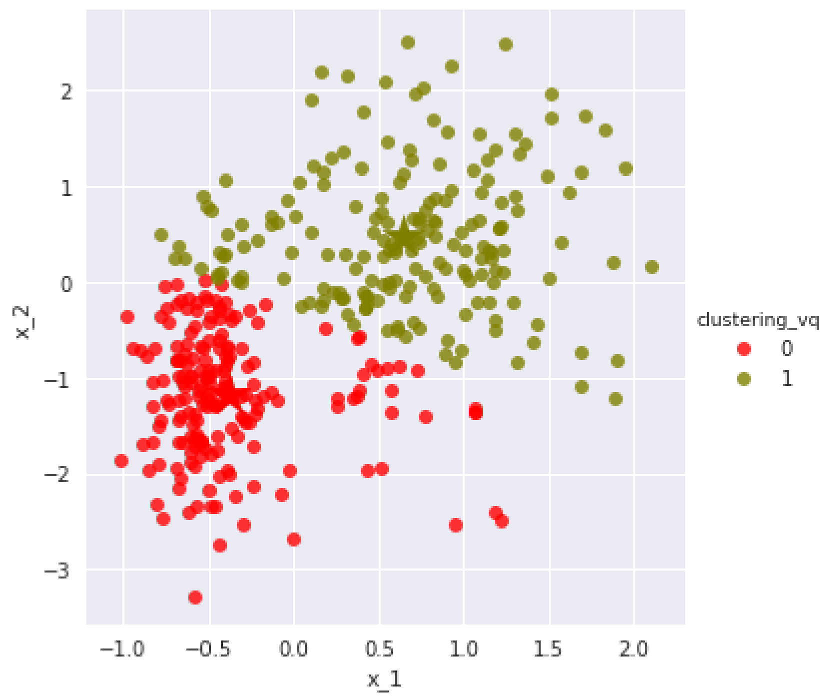
The preceding exercises walked you through how to implement k-means, but as you might have imagined, there are existing implementations a following shows you how to use Scipy's implementation, which should yield similar results. If you are asked to use k-means in a future lab (or ex use this one.

In [29]:
```
from scipy.cluster import vq
```

In [30]:
```
# `distortion` below is the similar to WCSS.
# It is called distortion in the Scipy documentation
# since clustering can be used in compression.
centers_vq, distortion_vq = vq.kmeans(points, k)

# vq return the clustering (assignment of group for each point)
# based on the centers obtained by the kmeans function.
# _ here means ignore the second return value
clustering_vq, _ = vq.vq(points, centers_vq)

print("Centers:\n", centers_vq)
print("\nCompare with your method:\n", centers, "\n")
print("Distortion (WCSS):", distortion_vq)

df['clustering_vq'] = clustering_vq
make_scatter_plot(df, hue='clustering_vq', centers=centers_vq)
```

```
n_matches_vq = count_matches(df['label'], df['clustering_vq'])
print(n_matches_vq,
      "matches out of",
      len(df), "possible",
      "(~ {:.1f}%)".format(100.0 * n_matches_vq / len(df)))
```

```
Centers:
 [[-0.37382602 -1.18565619]
 [ 0.64980076  0.4667703 ]]

Compare with your method:
 {0: array([202.52108949, 198.84707504, 192.62337998]), 1: array([134.23584777, 125.34568766, 101.93988
rray([38.7890798 , 45.35163548, 48.17869416])}

Distortion (WCSS): 0.7500461744207869
329 matches out of 375 possible (~ 87.7%)
```



Fin! That marks the end of this notebook. Don't forget to submit it!

< Previous

Next Up: Topic 15: Compression via PCA 21 min >

edX

[About](#)
[Affiliates](#)
[edX for Business](#)
[Open edX](#)
[Careers](#)
[News](#)

Legal

[Terms of Service & Honor Code](#)

[Privacy Policy](#)
[Accessibility Policy](#)
[Trademark Policy](#)
[Sitemap](#)

Connect

[Blog](#)
[Contact Us](#)
[Help Center](#)
[Media Kit](#)
[Donate](#)

GET IT ON Google play
Download on the App Store