lab6

October 5, 2016

1 EE 379K: Lab 6

1.1 Rohan Nagar and Wenyang Fu

In [1]:
```
import random
import numpy as np
import pandas as pd
import matplotlib as mpl
import matplotlib.pyplot as plt

from mpl_toolkits.mplot3d import Axes3D

%matplotlib inline
%load_ext autoreload
%autoreload 2
```

2 Problem 1: K-Means

2.0.1 Part 1

Implement the K-Means algorithm with a random initalization. Recall that there are two steps: given the centers, computing the clusters, and given the clusters, computing the centers. As discussed in class, the second is straightforward. Describe how you do the first step.

2.0.2 Answer

You can do the first step with the following basic algorithm. You have a list of the centers and a list of all the points that need to be put in a cluster.

For each point, compute the distance (using the same distance metric you have been using) from the point to each center. Choose the distance with the smallest magnitude. All of the points with that center as the closest will form a cluster around that center.

In [2]:
```
# %load kmeans.py
import numpy as np
from collections import defaultdict
from copy import deepcopy


class KMeans(object):
    def __init__(self, df, k):
        """
        Given a pandas dataframe and the of clusters,
        Initializes k-means to its initial K centroids
```

```
        via uniform random sampling (without replacement).

        Params:
            df - pandas dataframe containing points to be clustered
            k - number of clusters to form
        """

        # Got some help/inspiration from this StackOverflow link:
        # http://stackoverflow.com/questions/5466323/how-exactly-does-k-means-work
        # Also read this link (a bit confusing, but helpful too):
        # https://datasciencelab.wordpress.com/2014/01/15/improved-seeding-for-clustering-with-

        self.k = np.min(k, df.shape[0])
        self.points = df.as_matrix()
        self.centers = df.sample(self._k).as_matrix()
        self.clusters = defaultdict(list)

    def _find_center(self, p):
        """ Calculates the closest center for a point based
        on Euclidean distance.

        Args:
            p - a point within self.points

        Returns:
            Index of the center closest to current pixel
        """
        center_dists = np.linalg.norm(p - self.centers, axis=1)
        return np.argmin(center_dists)

    def _assign(self):
        """Assign each point to its nearest centroid."""
        clusters = defaultdict(list)
        for p in self.points:
            center_idx = self._find_center(p)
            clusters[center_idx].append(p)
        self.clusters = clusters

    def update_centroids(self):
        """ Update centroids for a single iteration of k-means."""
        self._assign()
        new_centers = np.zeros(self.centers.shape)
        # Recompute new centroids
        for center_idx, cluster in sorted(self.clusters).items():
            # transform list of points in cluster -> ndarray of points
            cluster_pts = np.array(cluster)
            # Take the average of all points (aka along the rows, axis=0)
            # associated with the current centroid, and
            # use that as the new centroid.
            avg = np.sum(cluster_pts, axis=0) / cluster_pts.shape[0]
            new_centers[center_idx] = avg
        self.centers = new_centers

    def run(self, num_iters=10):
```

```
        """
        Runs K-means++ for num_iters iterations, or until
        centroids converge.
        """
        for i in range(num_iters):
            old_centroids = deepcopy(self.centers)
            self.update_centroids()
            if i != 0:
                if self.centers == old_centroids:
                    break
        return zip(self.centers, list(self.clusters.values()))
```

2.0.3 Part 2

Now implement K-Means++, i.e., the K-Means++ initalization.

In [3]:
```
# %load kmeanspp.py
import numpy as np
from collections import defaultdict
from copy import deepcopy


class KMeansPP(object):
    def __init__(self, df, k):
        """
        Given a pandas dataframe and the of clusters,
        Initializes k-means to its initial K centroids
        via k-means++ init

        Params:
            df - pandas dataframe containing points to be clustered
            k - number of clusters to form
        """

        # Got some help/inspiration from this StackOverflow link:
        # http://stackoverflow.com/questions/5466323/how-exactly-does-k-means-work
        # Also read this link (a bit confusing, but helpful too):
        # https://datasciencelab.wordpress.com/2014/01/15/improved-seeding-for-clustering-with-

        # If the number of points is less than the number
        # of clusters, then set k=number of points
        self.k = min(k, df.shape[0])
        self.points = df.as_matrix()
        self.centers = []

        # Choose initial center uniformly at random from the points.
        c1_index = np.random.randint(low=0, high=self.points.shape[0])
        # self.centers[0] = self.points[c1_index]
        self.centers.append(self.points[c1_index])

        for i in range(1, k):
            # For each iteration, Compute the vector containing the square
            # distances between all points in the dataset

            dist_vec = np.array([min([(c - x) @ (c - x)
```

```
                                for c in self.centers])
                            for x in self.points])
        # choose each subsequent center from self.points,
        # randomly drawn from the normalized probability distribution
        # over dist_vec.
        probs = dist_vec / dist_vec.sum()
        cumprobs = probs.cumsum()
        r = np.random.rand()

        for j, p in enumerate(cumprobs):
            if r < p:
                ci_index = j  # Index of every subsequent centroid
                break
        self.centers.append(self.points[ci_index])

    self.centers = np.array(self.centers)
    self.clusters = defaultdict(list)

def _find_center(self, p):
    """ Calculates the closest center for a point based
    on Euclidean distance.

    Args:
        p - a point within self.points

    Returns:
        Index of the center closest to current pixel
    """
    center_dists = np.linalg.norm(p - self.centers, axis=1)
    return np.argmin(center_dists)

def _assign(self):
    """Assign each point to its nearest centroid."""
    clusters = defaultdict(list)
    for p in self.points:
        center_idx = self._find_center(p)
        clusters[center_idx].append(p)
    self.clusters = clusters

def _update_centroids(self):
    """ Update centroids for a single iteration of k-means."""
    self._assign()
    new_centers = np.zeros(self.centers.shape)
    # Recompute new centroids
    for center_idx, cluster in sorted(self.clusters.items()):
        # transform list of points in cluster -> ndarray of points
        cluster_pts = np.array(cluster)
        # Take the average of all points (aka along the rows, axis=0)
        # associated with the current centroid, and
        # use that as the new centroid.
        avg = np.sum(cluster_pts, axis=0) / cluster_pts.shape[0]
        new_centers[center_idx] = avg
    self.centers = new_centers
```

```
    def run(self, num_iters=10):
        """
        Runs K-means++ for num_iters iterations, or until
        centroids converge.
        """
        for i in range(num_iters):
            old_centroids = deepcopy(self.centers)
            self._update_centroids()
            if i != 0:
                if self.centers == old_centroids:
                    break
        return zip(self.centers, list(self.clusters.values()))
```

2.0.4 Part 3

Download the data `iris_data.csv` from Canvas. Plot Sepal Width vs. Sepal Length.

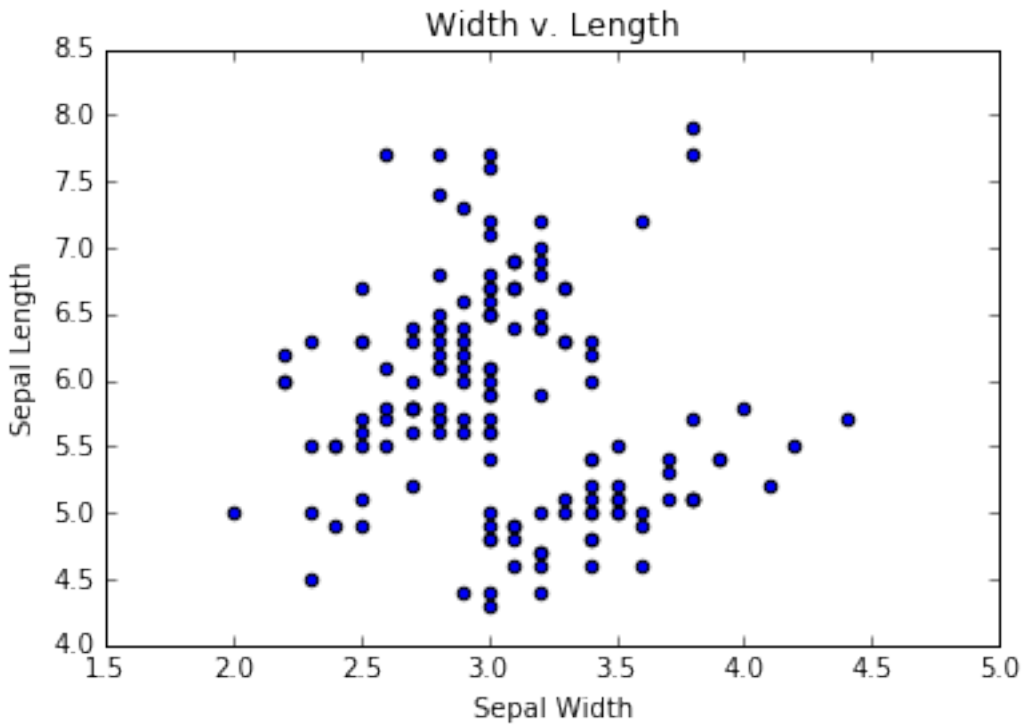
```
In [4]: iris = pd.read_csv('iris_data.csv')

        iris.head()
```

Out[4]:

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
1	5.1	3.5	1.4	0.2	setosa
2	4.9	3.0	1.4	0.2	setosa
3	4.7	3.2	1.3	0.2	setosa
4	4.6	3.1	1.5	0.2	setosa
5	5.0	3.6	1.4	0.2	setosa

```
In [5]: plt.scatter(iris['Sepal.Width'], iris['Sepal.Length'])
        plt.title('Width v. Length')
        plt.xlabel('Sepal Width')
        plt.ylabel('Sepal Length')
```

```
Out[5]: <matplotlib.text.Text at 0x10957db70>
```




```
In [6]: iris_X = iris.drop(['Species', 'Petal.Length', 'Petal.Width'], axis=1)
        iris_X.head(5)
```

Out[6]:

	Sepal.Length	Sepal.Width
1	5.1	3.5
2	4.9	3.0
3	4.7	3.2
4	4.6	3.1
5	5.0	3.6

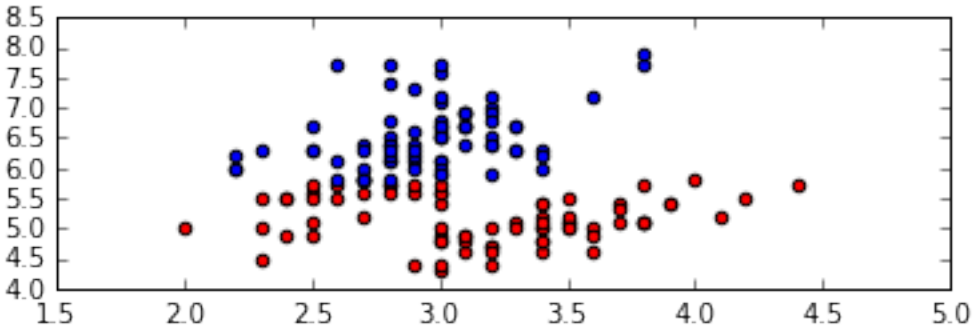
2.0.5 Part 4

Use your K-Means++ algorithm to cluster the above two variables into 2, 3, 4, and 5 clusters.

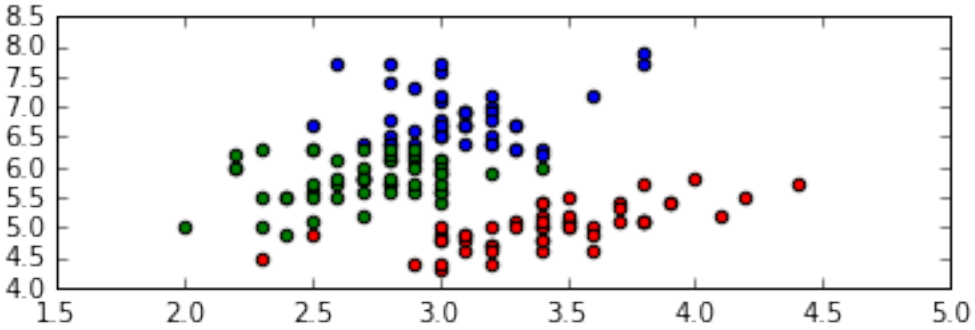
```
In [7]: def show_clusters(clusters, reverse=False):
            fig = plt.figure()
            colors = ['r', 'b', 'g', 'm', 'y']
            for i, cluster in enumerate(list(clusters)):
                _, points = cluster
                # Seperate into x and y
                if not reverse:
                    x = [point[1] for point in points]
                    y = [point[0] for point in points]
                else:
                    x = [point[0] for point in points]
                    y = [point[1] for point in points]

                ax1 = fig.add_subplot(2, 1, 1)
```

```
            ax1.scatter(x, y, c=colors[i])

        plt.show()
```

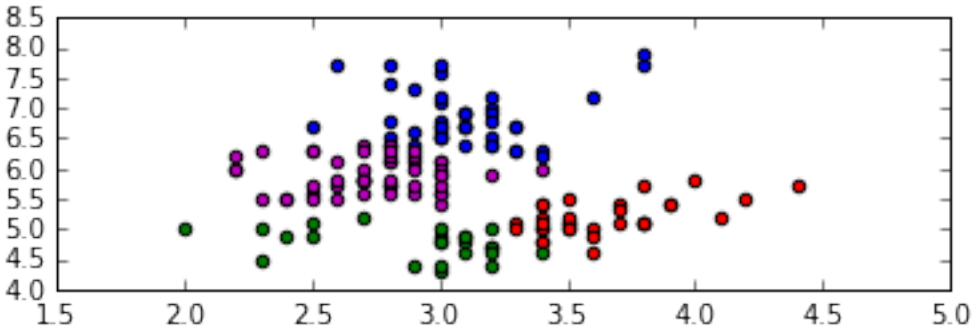
In [8]:
```
from kmeanspp import KMeansPP

model = KMeansPP(iris_X, 2)
clusters_2 = model.run(num_iters=25)
show_clusters(clusters_2)
```



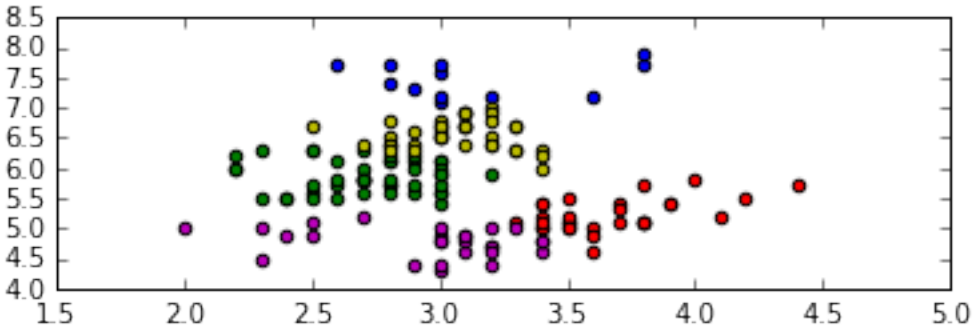
In [9]:
```
model = KMeansPP(iris_X, 3)
clusters_3 = model.run(num_iters=25)
show_clusters(clusters_3)
```



In [10]:
```
model = KMeansPP(iris_X, 4)
clusters_4 = model.run(num_iters=25)
show_clusters(clusters_4)
```



In [11]:
```
model = KMeansPP(iris_X, 5)
clusters_5 = model.run(num_iters=25)
show_clusters(clusters_5)
```



3 Problem 2: Spectral Clustering

Spectral clustering, as described in class, is a powerful and much-used algorithm. A very in-depth tutorial can be found here: http://www.cs.cmu.edu/~aarti/Class/10701/readings/Luxburg06_TR.pdf. In class we discussed the application of spectral clustering to a given graph. But we can also apply it to other data sets, by turning them into weighted graphs. This can be done in several ways, but roughly speaking, we convert by means of a similarity function: two nodes have an edge between them with weight equal to their similiarity. Read section 2 of the above tutorial on different ways to create the similiarity graph. Then read section 4 on the different graph clustering algorithms, especically focusing on the one by Ng, Jordan and Weiss (2002). This is the one we discussed in class.

3.0.1 Part 1

Implement the Ng, Jordan and Weiss spectral clustering algorithm. You may use basic linear algebra functions, including SVD and eigenvalue decomposition, but do not use `sklearn.cluster.spectral.clustering` or anything similar. Of course, you can use your K-Means algorithm from above for the last step.

In [12]:
```
# %load spectral_clustering.py
import numpy as np
```

```
import pandas as pd
from numpy import linalg as LA
from collections import defaultdict
from copy import deepcopy


class SpectralCluster(object):
    def __init__(self, df, k, sigma=1):
        """
        Given a pandas dataframe and the of clusters,
        Initializes spectral clustering to its initial K centroids
        via uniform random sampling (without replacement).

        Params:
            df - pandas dataframe containing points to be clustered
            k - number of clusters to form
            sigma - scaling parameter.
        """

        # Implemented against the following paper from 2001:
        # http://ai.stanford.edu/~ang/papers/nips01-spectral.pdf

        self.k = min(k, df.shape[0])
        self.orig_points = df.as_matrix()
        self.centers = None  # Centroids in spectral subspace
        self.points = None  # Points in spectral subspace

        self._spectral_init(sigma)
        self.clusters = defaultdict(list)

    def _spectral_init(self, sigma):
        """ Map the original dataset onto the spectral
        subspace and find the initial centroids in the subspace.
        """
        # Form the affinity matrix A
        n = self.orig_points.shape[0]  # Number of points.
        points = self.orig_points
        A = np.zeros((n, n))
        for i in range(n):
            A[i, :] = LA.norm(points[i] - points, axis=1) ** 2
        A = np.exp(-A / (2 * np.power(sigma, 2)))
        np.fill_diagonal(A, 0)

        # Find the Laplacian matrix L, which is equal to D^(-1/2)AD^(-1/2),
        # where D is defined as a diagonal matrix with diagonal entries equal
        # to the sums of the rows of A.
        D_sqrt_recip = np.diag(np.reciprocal(
            np.power(A.sum(axis=1), .5)))  # D^(-1/2)
        L = D_sqrt_recip @ A @ D_sqrt_recip
        w, v = LA.eig(L)  # Eigenvalues and eigenvectors of L
        eig_pairs = list(zip(w, v))

        # Sort eigenvalue/eigenvector pairs from high to low
        # eigenvecs should be orthogonal because L should be a symmetric matrix
```

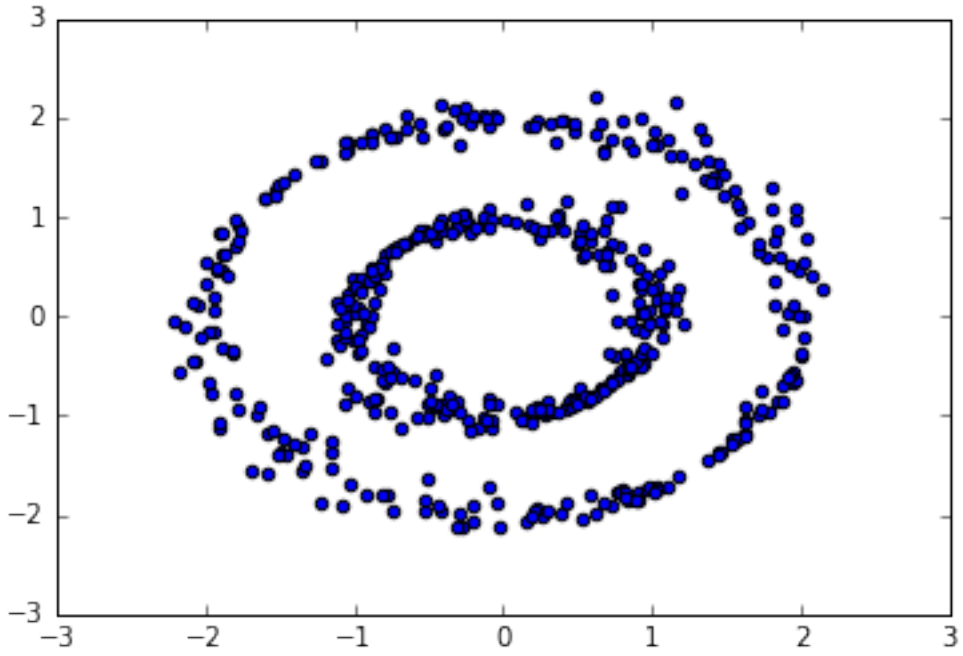
```
        # spectral clustering assumes this to be true.
        eig_pairs.sort(key=lambda x: x[0], reverse=True)
        # only keep k largest eigenvectors
        k_largest_w = [v for w, v in eig_pairs[:self.k]]
        # Convert list into a ndarray, where eigenvectors are the column vecs
        # of the matrix sorted greatest->least as column index increases
        # (left->right)
        X = np.array(k_largest_w).T

        # Form the matrix Y from X by renormalizing each of X's rows to have
        # unit length.
        Y = X / np.sqrt((X ** 2).sum(axis=1)).reshape(-1, 1)
        self.points = Y
        self.centers = pd.DataFrame(Y).sample(self._k).as_matrix()

    def _find_center(self, p):
        """ Calculates the closest center for a point based
        on Euclidean distance.

        Args:
            p - index of a point within self.points

        Returns:
            Index of the center closest to current pixel
        """
        center_dists = LA.norm(self.points[p] - self.centers, axis=1)
        return np.argmin(center_dists)

    def _assign(self):
        """Assign each point to its nearest centroid."""
        clusters = defaultdict(list)
        for p_idx, p in enumerate(self.points):
            center_idx = self._find_center(p)
            clusters[center_idx].append(p_idx)
        self.clusters = clusters

    def update_centroids(self):
        """ Update centroids for a single iteration of k-means."""
        self._assign()
        new_centers = np.zeros(self.centers.shape)
        # Recompute new centroids
        for center_idx, cluster in sorted(self.clusters).items():
            # transform list of point indices in cluster -> ndarray of points
            cluster_pts = np.array([self.points[p_idx] for p_idx in cluster])
            # Take the average of all points (aka along the rows, axis=0)
            # associated with the current centroid, and
            # use that as the new centroid.
            avg = np.sum(cluster_pts, axis=0) / cluster_pts.shape[0]
            new_centers[center_idx] = avg
        self.centers = new_centers

    def _spectral_to_original(self):
        """ Map the spectral clustering back to the original
        data points.
```

```
            Return:
                Clustering of points in original space.
            """
            # embedding is a nested list containing the indices of the points
            # that belong to each cluster.
            embedding = self.clusters.values()
            orig_clustering = [[self.orig_points[p_idx] for p_idx in cluster]
                               for cluster in embedding]
            return orig_clustering

        def run(self, num_iters=10):
            """
            Runs K-means++ for num_iters iterations, or until
            centroids converge.
            """
            for i in range(num_iters):
                old_centroids = deepcopy(self.centers)
                self.update_centroids()
                if i != 0:
                    if self.centers == old_centroids:
                        break
            # Map clustering in spectral subspace back to original dataset
            return zip(self.centers, self._spectral_to_original())
```

3.0.2 Part 2

Write a program that generates N data points uniformly at random on two concentric rings, and then add two-dimensional Gaussian noise of variance σ_1^2 and σ_2^2 to each point. There are four parameters of interest here: the number of points, the difference in the radii of the two rings, the amount (the variance) of noise added to the internal ring, and the amount of noise added to the external ring.

```
In [20]: def generate_data(n, inner_radii, outer_radii, sigma_one, sigma_two):
             points = []
             for i in range(n):
                 theta = np.random.uniform(0, 2 * np.pi)
                 if np.random.random_integers(1, 2) == 2:
                     radius = inner_radii
                     noise = np.random.normal(scale=sigma_one)
                 else:
                     radius = outer_radii
                     noise = np.random.normal(scale=sigma_two)

                 x = radius*np.cos(theta) + noise
                 y = radius*np.sin(theta) + noise

                 points.append([x, y])

             return points

         example_df = pd.DataFrame(generate_data(500, 1, 2, 0.1, 0.1))
         plt.scatter(example_df[0], example_df[1])

Out[20]: <matplotlib.collections.PathCollection at 0x10e133a58>
```



3.0.3 Part 3

Experiement with the 3 methods of creating similarity functions described in section 2 of the paper. Pick one that you like (clearly state which one), and fixing $N = 500$ and $\sigma_1 = \sigma_2 = 1$, and also the radius of the inner ring to be 1, find the range of the outer ring radius for which spectral clustering, using your chosen similarity function, succeeds in clustering the two different rings. Note that you will be using $k = 2$ in your spectral clustering algorithm.

Let's play with the 3 methods.

In [18]:
```
data = generate_data(500, 1, 3, 1, 1)

# e-neighborhood
e = 1.2
e_sim_matrix = []
for point in data:
    row = []
    for second_point in data:
        sim = np.linalg.norm(np.array(point)-np.array(second_point))
        if sim < e:
            row.append(1)
        else:
            row.append(0)

    e_sim_matrix.append(row)

# knn
k = 5
knn_sim_matrix = []
for point in data:
```

```
            sim = []
            for second_point in data:
                sim.append(np.linalg.norm(np.array(point)-np.array(second_point)))

            idx = np.argpartition(sim, k)
            idx = idx[:k]

            row = []
            for i in range(len(data)):
                if i in idx:
                    row.append(sim[i])
                else:
                    row.append(0)

            knn_sim_matrix.append(row)

        # fully connected
        sigma = 3
        full_sim_matrix = []
        for point in data:
            row = []
            for second_point in data:
                # Gaussian similarity
                sim = np.exp(-(np.linalg.norm(np.array(point)-np.array(second_point))**2)/(2*sigma**2))
                row.append(sim)

            full_sim_matrix.append(row)
```

We like the K-Nearest-Neighbors method for a similarity graph. This is because it is much more sparse and so easier to compute e-vects for. It also is able to be very similar to points near it, which is good.

4 Problem 3: Spectral Clustering vs. K-Means

4.0.1 Part 1

Load the data from the two-dimenstional `two_moons.csv` and the three-dimensional `linked_chains.csv`. Each row represents a point. Note that the first column gives the ground truth - the cluster identity of each point - and so can be ignored. Plot each in 2d and 3d respectively, so you can see what is going on.

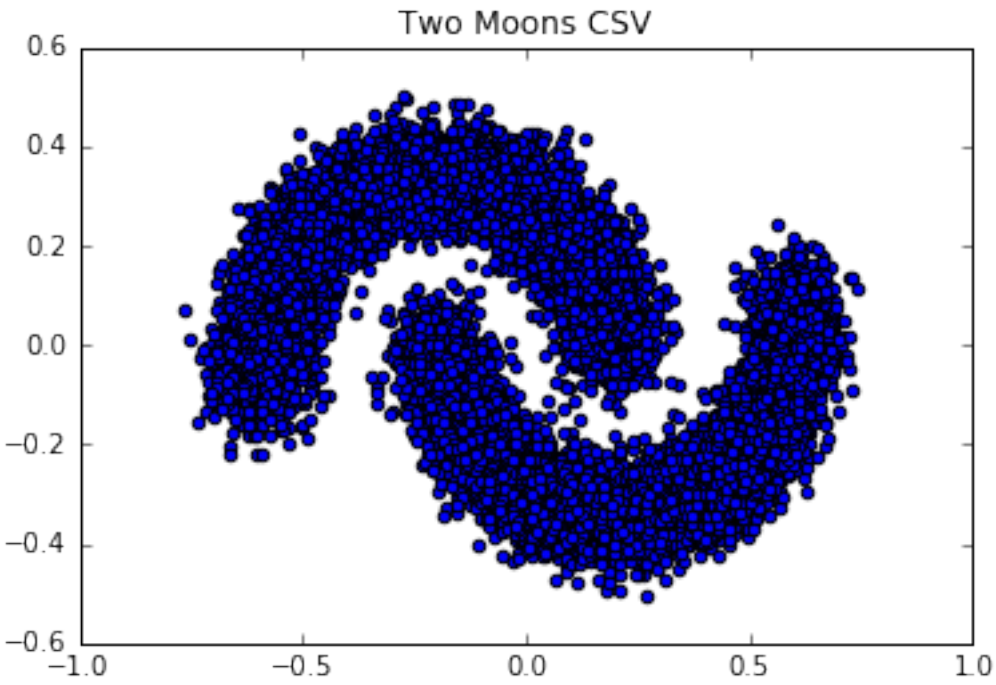
```
In [14]: moons = pd.read_csv('two_moons.csv', header=None)
         moons.head()
```

Out[14]:

	0	1	2
0	2.0	0.250844	-0.001121
1	2.0	0.221185	0.008294
2	2.0	0.257351	-0.052437
3	2.0	0.197335	-0.030503
4	2.0	0.256422	-0.019958

```
In [15]: plt.scatter(moons[1], moons[2])
         plt.title('Two Moons CSV')
```

```
Out[15]: <matplotlib.text.Text at 0x7f44071ac470>
```



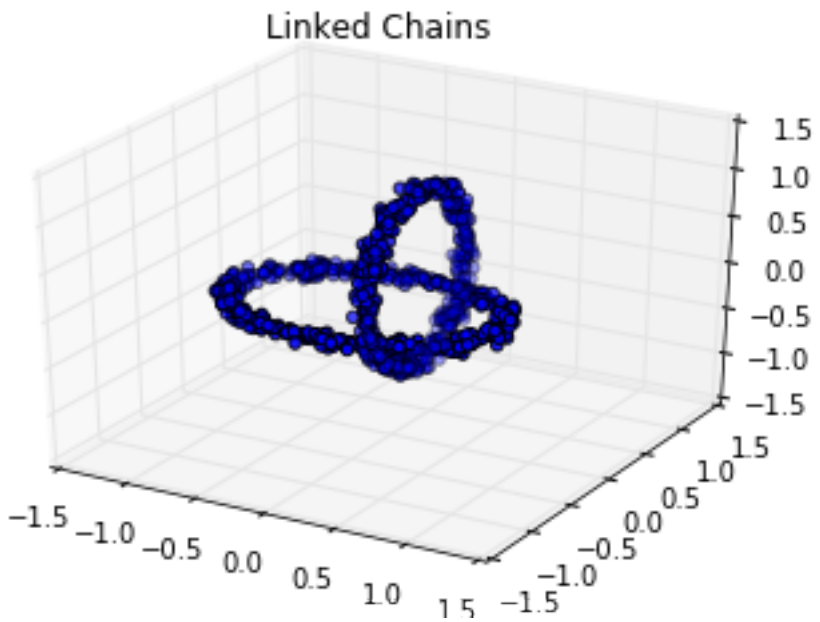

```
In [16]: chains = pd.read_csv('linked_chains.csv', header=None)
         chains.head()
```

Out[16]:

	0	1	2	3
0	1.0	-0.424914	-0.879778	0.019944
1	1.0	0.609164	-0.851496	0.018148
2	1.0	0.370662	0.258447	-0.002878
3	1.0	0.087514	0.347741	0.004298
4	1.0	0.488615	-0.911575	0.033750

```
In [17]: fig = plt.figure()
         ax = fig.add_subplot(111, projection='3d')
         ax.scatter(chains[1], chains[2], chains[3], label='Linked Chains')
         plt.title('Linked Chains')
         fig.show()
```

```
/home/aetherzephyr/anaconda3/lib/python3.5/site-packages/matplotlib/figure.py:397: UserWarning: matplotl
  "matplotlib is currently using a non-GUI backend, "
```



4.0.2 Part 2

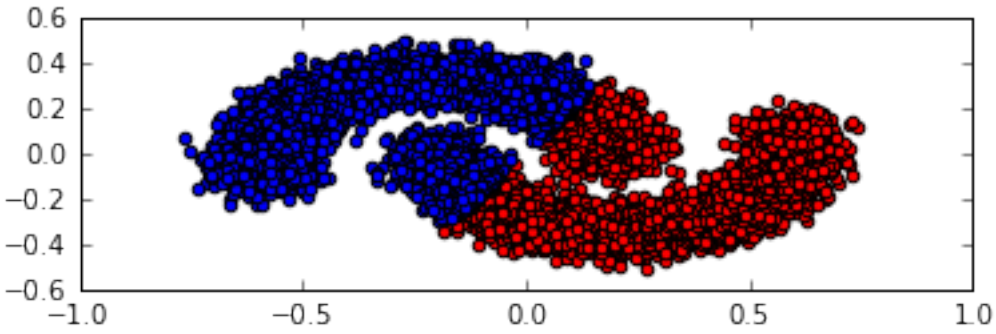
Run K-Means on both. How do you do?

```
In [18]: moons_X = moons.drop(0, axis=1)
         moons_X.head()
```

Out[18]:

	1	2
0	0.250844	-0.001121
1	0.221185	0.008294
2	0.257351	-0.052437
3	0.197335	-0.030503
4	0.256422	-0.019958

```
In [19]: model = KMeansPP(moons_X, 2)
         moon_clusters = model.run(num_iters=20)
         show_clusters(moon_clusters, reverse=True)
```



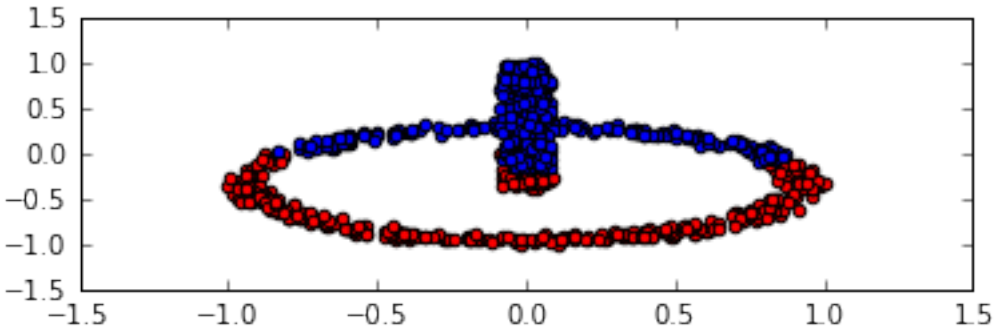
Clearly, K-Means does not do a very good job clustering the Moons dataset.

```
In [20]: chains_X = chains.drop(0, axis=1)
         chains_X.head()
```

Out[20]:

	1	2	3
0	-0.424914	-0.879778	0.019944
1	0.609164	-0.851496	0.018148
2	0.370662	0.258447	-0.002878
3	0.087514	0.347741	0.004298
4	0.488615	-0.911575	0.033750

```
In [21]: model = KMeansPP(chains_X, 2)
         chain_clusters = model.run(num_iters=20)
         show_clusters(chain_clusters, reverse=True)
```



Again, based on this plot we can see that K-Means doesn't do a good job of clustering the chains. The clusters were done in 3D but the plot is shown in 2D.

4.0.3 Part 3

Find the best similarity function you can, and run Spectral Clustering. Which is better?

```
In [25]: from sklearn.metrics import pairwise_distances

         A = pairwise_distances(moons_X.as_matrix(), moons_X.as_matrix())
         A = np.exp(-A / (2 * np.power(1, 2)))
         np.fill_diagonal(A, 0)

         # Find the Laplacian matrix L, which is equal to D^(-1/2)AD^(-1/2),
         # where D is defined as a diagonal matrix with diagonal entries equal
         # to the sums of the rows of A.
         D_sqrt_recip = np.diag(np.reciprocal(
             np.sqrt(A.sum(axis=1))))  # D^(-1/2)
         L = D_sqrt_recip @ A @ D_sqrt_recip
```

```
In [ ]: import numpy.linalg as LA

        w, v = LA.eig(L)  # Eigenvalues and eigenvectors of L
        eig_pairs = list(zip(w, v))
```

```
In [ ]: from spectral_clustering import SpectralCluster
```

```
        model = SpectralCluster(moons_X, 2)
        moon_clusters_spectral = model.run(num_iters=1)
        show_clusters(moon_clusters_spectral, reverse=True)
```

In [ ]:
```
model = SpectralCluster(chains_X, 2)
chain_clusters_spectral = model.run()
show_clusters(chain_clusters_spectral, reverse=True)
```

4.1 We decided to implement spectral clustering using a Gaussian (rbf) kernel. We realized that the computation would take many hours due to the size of the dataset, and unfortuantely we don't have enough time to run it. However, we know that Spectral Clustering in these cases would work well. When the data is shaped like it is, Spectral Clustering tends to do a good job.

5 Problem 4: More Spectral Clustering vs. K-Means

5.0.1 Part 1

Generate an example where K-Means does better than Spectral Clustering. Explain how you constructed it and why, in addition to showing the plots.

5.0.2 K-means would do better than spectral clustering in situations where every cluster is linearly separable. This is because K-means greedily assigns points to clusters via Euclidean distance, while spectral clustering projects points to a lower-dimensional manifold, which would better suit clusters that are not linearly separable.

6 Problem 5: Computational Complexity and Spectral Clustering

6.0.1 Part 1

This is a more open ended question. Computing the eigenvalues and eigenvectors of a matrix is not easy, and the complexity depends on the size of the matrix among other factors. An important factor is the sparsity of the matrix. Which aspects of your algorithm for spectral clustering could affect this? Using one of the examples from above, or another example of your choosing, try to illustrate this pheonomeon.

6.0.2 The choice of kernel would affect the time complexity of the algorithm - some kernels are far easier to compute compared to others. In addition, the choice of algorithm for finding the eigenvectors will also have a big impact on the time complexity. There are some algorithms that are designed for sparse matricies and some that are designed for dense matricies. This is because they are so different and require different types of computation. Finding the e-vals and e-vects is a hard task, especially in dense matrices.