CODE APPENDIX

clustering_classes.py

```
"""
EECS 445 - Introduction to Machine Learning
Winter 2019 - Homework 3
Clustering Classes
"""

import numpy as np
from scipy import stats

class Point(object):
    """
    Represents a data point
    """

    def __init__(self, features, label=None):
        """
        Initialize label and attributes
        """
        self.features = features
        self.label = label

    def dimensionality(self):
        """Returns dimension of the point"""
        return len(self.features)

    def get_features(self):
        """Returns features"""
        return self.features

    def distance(self, other):
        """
        other: point, to which we are measuring distance to
        Return Euclidean distance of this point with other
        """
        # TODO: Implement this function
        return np.linalg.norm(self.features - other.features)


    def get_label(self):
        """Returns label"""
        return self.label

class Cluster(object):
    """
    A Cluster is defined as a set of elements
    """

    def __init__(self, points):
        """
        Elements of a cluster are saved in a list, self.points
        """
        self.points = points
```

```
    def get_points(self):
        """Returns points in the cluster as a list"""
        return self.points

    def get_label(self):
        """Returns label of the cluster, which is determined by the
            mode of labels"""
        labels = [point.get_label() for point in self.points]
        cluster_label, count = stats.mode(labels)
        return cluster_label[0]

    def get_purity(self):
        """Returns number of points in cluster and the number of points
            with the most common label"""
        labels = [point.get_label() for point in self.points]
        cluster_label, count = stats.mode(labels)
        return len(labels), np.float64(count)

    def get_centroid(self):
        """Returns centroid of the cluster"""
        # TODO: Implement this function
        if len(self.points) == 0: return Point(0)
        features = [point.get_features() for point in self.points]
        return Point(np.sum(features, axis = 0) / len(features))



    def equivalent(self, other):
        """
        other: Cluster, what we are comparing this Cluster to
        Returns true if both Clusters are equivalent, or false otherwise
        """
        if len(self.get_points()) != len(other.get_points()):
            return False
        matched = []
        for p1 in self.get_points():
            for point2 in other.get_points():
                if p1.distance(point2) == 0 and point2 not in matched:
                    matched.append(point2)
        return len(matched) == len(self.get_points())

class ClusterSet(object):
    """
    A ClusterSet is defined as a list of clusters
    """

    def __init__(self):
        """
        Initialize an empty set, without any clusters
        """
        self.clusters = []

    def add(self, c):
        """
        c: Cluster
        Appends a cluster c to the end of the cluster list
        only if it doesn't already exist in the ClusterSet.
```

```
        If it is already in self.clusters, raise a ValueError
        """
        if c in self.clusters:
            raise ValueError
        self.clusters.append(c)

    def get_clusters(self):
        """Returns clusters in the ClusterSet"""
        return self.clusters[:]

    def get_centroids(self):
        """Returns centroids of each cluster in the ClusterSet as a list"""
        # TODO: Implement this function

        centroids = [cluster.get_centroid() for cluster in self.clusters]
        return centroids

    def get_score(self):
        """
            Returns accuracy of the clustering given by the clusters
            in ClusterSet object
        """
        total_correct = 0
        total = 0
        for c in self.clusters:
            n, n_correct = c.get_purity()
            total = total + n
            total_correct = total_correct + n_correct

        return total_correct / float(total)

    def num_clusters(self):
        """Returns number of clusters in the ClusterSet"""
        return len(self.clusters)

    def equivalent(self, other):
        """
        other: another ClusterSet object
        Returns true if both ClusterSets are equivalent, or false otherwise
        """
        if len(self.get_clusters()) != len(other.get_clusters()):
            return False
        matched = []
        for c1 in self.get_clusters():
            for c2 in other.get_clusters():
                if c1.equivalent(c2) and c2 not in matched:
                    matched.append(c2)
        return len(matched) == len(self.get_clusters())
```

clustering.py

```
"""
EECS 445 - Introduction to Machine Learning
Winter 2019 - Homework 3
Clustering
"""
import numpy as np
import random
import matplotlib.pyplot as plt
from sklearn.cluster import SpectralClustering
from sklearn.metrics.pairwise import pairwise_distances
from operator import methodcaller

from clustering_classes import Cluster, ClusterSet, Point
from data.landmarks import LandmarksDataset
from utils import denormalize_image

def random_init(points, k):
    """
    Arguments:
        points: a list of point objects
        k: Number of initial centroids/medoids
    Returns:
        List of k unique points randomly selected from points
    """
    # TODO: Implement this function
    random_result = random.sample(range(0, len(points)), k)
    result = []
    for i in range(k) : result += [points[random_result[i]]]
    return result

def k_means_pp_init(points, k):
    """
    Arguments:
        points: a list of point objects
        k: Number of initial centroids/medoids
    Returns:
        List of k unique points selected from points
    """
    # TODO: Implement this function
    result = [points[random.randint(0, len(points) - 1)]]

    while len(result) != k:
        distance = np.array([])
        for point in points:
            min_dis = float("inf")
            for centroid in result: min_dis = min(min_dis, point.distance(centroid))
            distance = np.append(distance, min_dis ** 2)
        distance = distance / np.sum(distance)
        result += [np.random.choice(points, 1, p = distance)[0]]

    return result
```

```
def k_means(points, k, init='random'):
    """
    Clusters points into k clusters using k_means clustering.
    Arguments:
        points: a list of Point objects
        k: the number of clusters
        init: The method of initialization. One of ['random', 'kpp'].
              If init='kpp', use k_means_pp_init to initialize clusters.
              If init='random', use random_init to initialize clusters.
              Default value 'random'.
    Returns:
        Instance of ClusterSet with k clusters
    """
    # TODO: Implement this function
    centroids = []
    if init == 'random': centroids = random_init(points, k)
    if init == 'kpp': centroids = k_means_pp_init(points, k)
    points_cluster = [[] for i in range(k)]


    for n in range(len(points)):
        min_dis, min_cluster = float("inf"), 0
        for i in range(k):
            if min_dis > points[n].distance(centroids[i]):
                min_cluster = i
                min_dis = points[n].distance(centroids[i])
        points_cluster[min_cluster] += [points[n]]

    clusters = [Cluster(point) for point in points_cluster]
    Cluster_new = ClusterSet()
    for cluster in clusters: Cluster_new.add(cluster)
    Cluster_set = ClusterSet()

    while (not Cluster_new.equivalent(Cluster_set)):
        Cluster_set = Cluster_new
        centroids = Cluster_set.get_centroids()
        points_cluster = [[] for i in range(k)]

        for n in range(len(points)):
            min_dis, min_cluster = float("inf"), 0
            for i in range(k):
                if min_dis > points[n].distance(centroids[i]):
                    min_cluster = i
                    min_dis = points[n].distance(centroids[i])
            points_cluster[min_cluster] += [points[n]]

        clusters = [Cluster(point) for point in points_cluster]
        Cluster_new = ClusterSet()
        for cluster in clusters: Cluster_new.add(cluster)

    return Cluster_new



def spectral_clustering(points, k):
    """
    Uses sklearn's spectral clustering implementation to cluster the input
```

```
    data into k clusters
    Arguments:
        points: a list of Points objects
        k: the number of clusters
    Returns:
        Instance of ClusterSet with k clusters
    """
    X = np.array([point.get_features() for point in points])
    spectral = SpectralClustering(
        n_clusters=k, n_init=1, affinity='nearest_neighbors', n_neighbors=50)
    y_pred = spectral.fit_predict(X)
    clusters = ClusterSet()
    for i in range(k):
        cluster_members = [p for j, p in enumerate(points) if y_pred[j] == i]
        clusters.add(Cluster(cluster_members))
    return clusters

def plot_performance(k_means_scores, kpp_scores, spec_scores, k_vals):
    """
    Uses matplotlib to generate a graph of performance vs. k
    Arguments:
        k_means_scores: A list of len(k_vals) average purity scores from
            running the k-means algorithm with random initialization
        kpp_scores: A list of len(k_vals) average purity scores from running
            the k-means algorithm with k_means++ initialization
        spec_scores: A list of len(k_vals) average purity scores from running
            the spectral clustering algorithm
        k_vals: A list of integer k values used to calculate the above scores
    """
    # TODO: Implement this function
    plt.xlabel("k")
    plt.ylabel("Purity")
    plt.plot(k_vals, k_means_scores, label = "k-means", linestyle = '--')
    plt.plot(k_vals, kpp_scores, label = "k-means++", linestyle = '--')
    plt.plot(k_vals, spec_scores, label = "spectral", linestyle = '--')
    plt.legend()
    plt.show()


def get_data():
    """
    Retrieves the data to be used for the k-means clustering as a list of
    Point objects
    """
    landmarks = LandmarksDataset(num_classes=5)
    X, y = landmarks.get_batch('train', batch_size=400)
    X = X.reshape((len(X), -1))
    return [Point(image, label) for image, label in zip(X, y)]

def visualize_clusters(kmeans, kpp, spectral):
    """
    Uses matplotlib to generate plots of representative images for each
    of the clustering algorithm. In each image, every row is from the same
    cluster, and from leftmost image is the medoid. Intra-cluster distance
    increases as we go from left to right.
    Arguments:
        - kmeans, kpp, and spectral: ClusterSet instances
    """
```

```
    def get_medoid_and_neighbors(points, num=4):
        D = pairwise_distances([p.features for p in points])
        distances = D.mean(axis=0)
        return np.array(points)[np.argsort(distances)[:num]].tolist()

    names = ['k-means', 'k-means++', 'spectral']
    cluster_sets = [kmeans, kpp, spectral]
    clusters_s = [sorted(cs.get_clusters(),
                key=methodcaller('get_label')) for cs in cluster_sets]

    for i, clusters in enumerate(clusters_s):
        num = 4
        k = len(clusters)
        fig, axes = plt.subplots(nrows=k, ncols=num, figsize=(8,8))
        plt.suptitle(names[i])
        for j in range(k):
            pts = get_medoid_and_neighbors(clusters[j].get_points(), num)
            for n in range(len(pts)):
                axes[j,n].imshow(denormalize_image(
                np.reshape(pts[n].features,(32,32,3))), interpolation='bicubic')
            axes[j,0].set_ylabel(clusters[j].get_label())

        for ax in axes.flatten():
            ax.set_xticks([])
            ax.set_yticks([])

        plt.savefig('4j_clusters_viz_{}.png'.format(names[i]),
                dpi=200, bbox_inches='tight')

def main():
    points = get_data()
    # TODO: Implement this function
    # for 3.h and 3.i

    """ 3.j """
    # Display representative examples of each cluster for clustering algorithms
    k_final, p_final, s_final = np.zeros(10), np.zeros(10), np.zeros(10)
    k_vals = [i for i in range(1,11)]
    # np.random.seed(42)
    for i in range(10):
        kmeans = np.array([k_means(points, i, 'random').get_score() for i in k_vals])
        kpp = np.array([k_means(points, i, 'kpp').get_score() for i in k_vals])
        spectral = np.array([spectral_clustering(points, i).get_score() for i in k_vals])
        k_final += kmeans
        p_final += kpp
        s_final += spectral
    k_final = k_final / 10
    p_final = p_final / 10
    s_final = s_final / 10
    plot_performance(k_final, p_final, s_final, k_vals)


    k_final, p_final, s_final = [], [], []
    k_vals = [i for i in range(1,11)]
    for i in range(10):
        k_final += [k_means(points, 7, 'random').get_score()]
        p_final += [k_means(points, 6, 'kpp').get_score()]
        s_final += [spectral_clustering(points, 5).get_score()]
```

```python
    print(s_final)
    print("k_final: Ave: %.4f, MAX: %.4f, MIN: %.4f" % (np.mean(k_final), max(k_final), min(k_final)))
    print("p_final: Ave: %.4f, MAX: %.4f, MIN: %.4f" % (np.mean(p_final), max(p_final), min(p_final)))
    print("s_final: Ave: %.4f, MAX: %.4f, MIN: %.4f" % (np.mean(s_final), max(s_final), min(s_final)))


if __name__ == '__main__':
    main()
```