Sparsity (PCA and Compressive Sensing)

ECE 283 Hw4

June 6, 2020

Student: Ivan Arevalo
Email: ifa@ucsb.com

Department of Electrical and Computer Engineering, UCSB

1 PCA

Let us start by generating high dimensional data (d = 100) while keeping the effective dimension smaller than d by generating Gaussian mixture samples from six i.i.d quasi-orthogonal vectors $\boldsymbol{u}$ with the following probability,

$$P[u[i] = 0] = 2/3,\ P[u[i] = +1] = 1/6,\ P[u[i] = -1] = 1/6 \tag{1}$$

N d-dimensional generated samples will be drawn equiprobably from 3 components described as follows,

Component 1: Generate $\mathbf{X} = \mathbf{u}_1 + V_1\mathbf{u}_2 + V_2\mathbf{u}_3 + \mathbf{N}$.
Component 2: Generate $\mathbf{X} = 2\mathbf{u}_4 + \sqrt{2}V_1\mathbf{u}_5 + V_2\mathbf{u}_6 + \mathbf{N}$.
Component 3: Generate $\mathbf{X} = \sqrt{2}\mathbf{u}_6 + V_1(\mathbf{u}_1 + \mathbf{u}_2) + \frac{1}{\sqrt{2}}V_2\mathbf{u}_5 + \mathbf{N}$.

Figure 1: D-dim Gaussian mixture components

where $\boldsymbol{V1}$ and $\boldsymbol{V_2}$ are coefficients drawn from a standard Gaussian distribution and $\boldsymbol{N}$ is a noise vector $\boldsymbol{N} \sim N(0, \sigma^2 \boldsymbol{I}_d)$ with $\sigma^2 = 0.01$.

We now do an SVD of the N d data matrix and analyze how many dominant singular values there as we increase N.

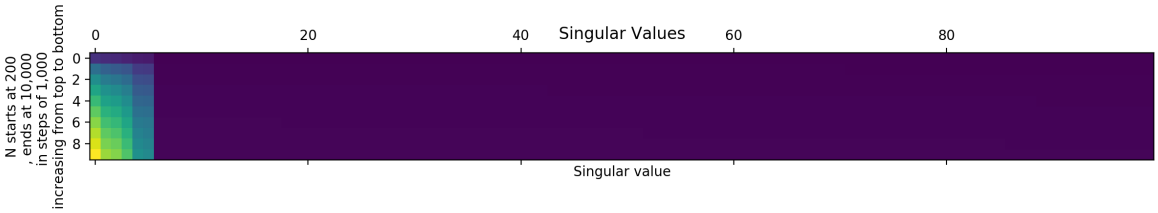

Figure 2: Singular Values as a function of N

We can see in figure 2 that as N increases, the number of dominant singular values stays the same but the most dominant singular value increases in relation to other dominant values. Looking at the bottom left corner, this is the dominant singular value for N = 10,000 and is significantly larger than the other singular values from the same matrix relative to those for N = 200. We can clearly see this by the difference in color intensity from the first and last row in figure 2. We can clearly see that for every iteration as N gets bigger, there are still 6 dominant singular values.

	1	2
1	0.0	1.0
2	1.0	0.0
3	0.0	1.0

	1	2	3
1	0.0	0.9396654719235364	0.06033452807646356
2	0.9959016393442623	0.0	0.004098360655737705
3	0.0	0.4289245982694685	0.5710754017305315

	1	2	3	4
1	0.0	0.06212664277180406	0.937873357228196	0.0
2	0.4783372365339578	0.000585480093676815	0.0	0.5210772833723654
3	0.0	0.569221260815822	0.430778739184178	0.0

	1	2	3	4	5
1	0.0	0.0	0.3255675029868578	0.6744324970131422	0.0
2	0.00351288056206089	0.515807962529274	0.0	0.0	0.4806791569086651
3	0.588380716934487	0.0	0.411619283065513	0.0	0.0

Figure 3: Empirical probabilities (low: blue, high: red) of how 5000 samples from each components (L = 1,2,3) map to the learned clusters (k = 1, .., K) for iterations with K = 2,3,4,5 (top to bottom)

We now perform a PCA and project the data down to the dominant d0 components to obtain an N d0 data matrix. Afterwards we implement K-means algorithm with different values of K = 2, 3, 4, 5. For each K we use K-means++ initializations and plot figure 3, the empirical probabilities indicating how the "ground truth" components map to the clusters learned.

Given that the data samples were generated from six quasi-orthogonal vectors, we now attempt to get some geometric insight between these and the cluster centers found by K-means. In order to so, we compute the normalized cross-correlation between vectors and compare how aligned they are.

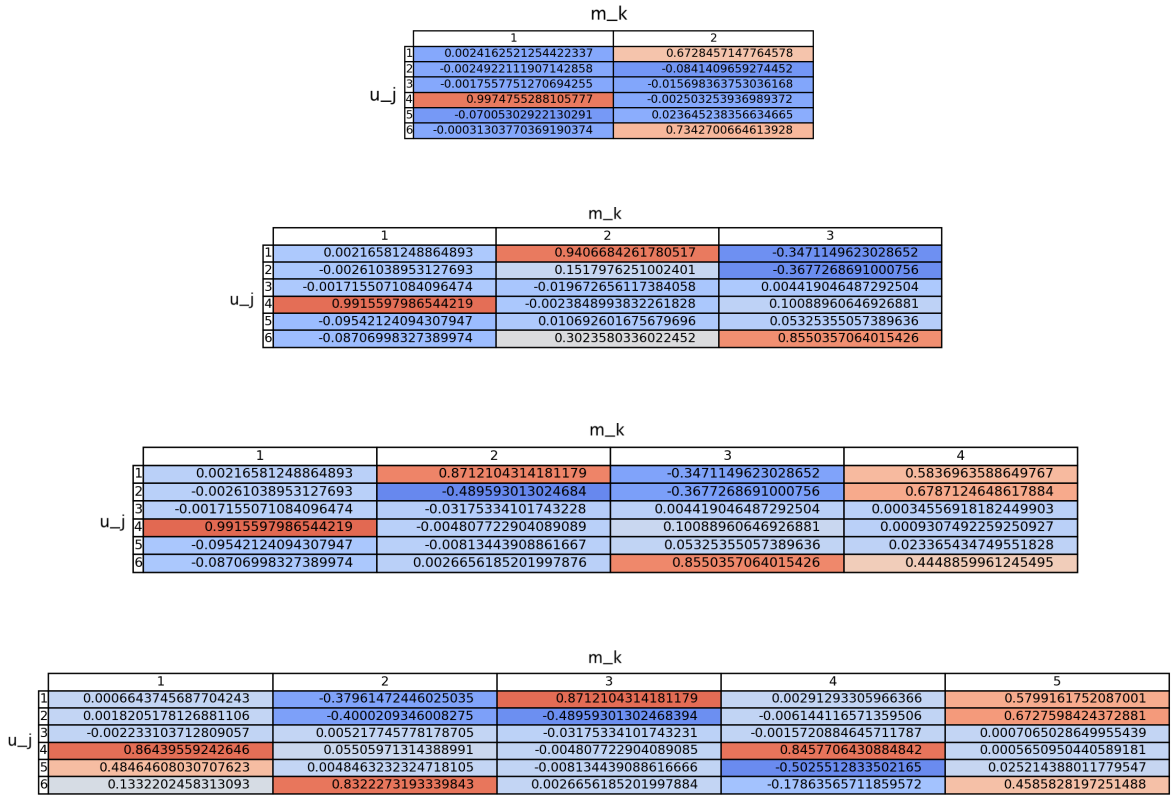

Figure 4: Cross correlation (low: blue, high: red) between the sample basis ($\boldsymbol{u_j}$ = 1,...,6) and the K-means-learned cluster means m_k (k = 1, .., K) for each iteration of K = 2, 3, 4, 5 (top to bottom)

Figure 4 shows that most learned cluster means are highly correlated with exactly one of the d0-dimensional projections of the vectors uj in the model. It is less so the case when running K-means with K (number of clusters to be learned) higher than L (number of components from which data originated). For K = 3 (second top down) in figure 4, each cluster mean is highly aligned with one of the six basis vectors. For high dimensional data, it finds a vector that best represents or aligns with the samples in that "direction" and groups them together. We have empirically shown that these d0-dimensional projections are highly correlated with the learned means.

2 Random Projections for Compressive Sensing

Now we will generate a m d matrix with i.i.d. entries drawn as follows:

$$P[\Phi_{ij} = 1] = 1/2,\ P[\Phi_{ij} = -1] = 1/2,\ P[u[i] = -1] = 1/6 \tag{2}$$

N d-dimensional generated samples will be drawn equiprobably from 3 components described as follows,

Component 1: Generate $\mathbf{x}_s = \mathbf{u}_1 + V_1\mathbf{u}_2 + V_2\mathbf{u}_3$.
Component 2: Generate $\mathbf{x}_s = 2\mathbf{u}_4 + \sqrt{2}V_1\mathbf{u}_5 + V_2\mathbf{u}_6$.
Component 3: Generate $\mathbf{x}_s = \sqrt{2}\mathbf{u}_6 + V_1(\mathbf{u}_1 + \mathbf{u}_2) + \frac{1}{\sqrt{2}}V_2\mathbf{u}_5$.

Figure 5: D-dim Gaussian mixture components

Next, we will compute the compressive projection:

$$\mathbf{p} = \frac{1}{\sqrt{m}}\Phi\mathbf{x}$$

and the define the following basis for the signal:

$$\mathbf{B} = [\mathbf{u}_1, \mathbf{u}_2, \mathbf{u}_3, \mathbf{u}_4, \mathbf{u}_5, \mathbf{u}_6]$$

Now we can find a sparse reconstruction of x_s based on y by solving the following lasso problem:

$$\hat{\mathbf{s}} = \arg\min_{\mathbf{s}} \tfrac{1}{2}||\mathbf{p} - \tfrac{1}{\sqrt{m}}\Phi\mathbf{B}\mathbf{s}||_2^2 + \lambda||\mathbf{s}||_1$$

$$\hat{\mathbf{x}_s} = \mathbf{B}\hat{\mathbf{s}}$$

From different reconstruction trial I found m = 100 to be satisfactory.

3 Code

```
import sample_generation
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import multivariate_normal
import pprint
import math


class K_means():

    def __init__(self, num_clusters, samples):
        self.num_clusters = num_clusters
        self.samples = samples
        self.one_hot_assignments = []
        self.cluster_means = []
        self.num_iterations = 0
        self.distortion = None
        self.jth_center_probabilities =
            np.ones(self.samples.shape[0])/self.samples.shape[0]

    def update_cluster_centers(self, initial_assignment=False, k_meansplusplus=False):

        if initial_assignment: # No seed provided, each run has a random cluster
            initialization
            np.random.seed()
            while len(self.cluster_means) < self.num_clusters:
                index = np.random.choice(np.arange(0, np.shape(self.samples)[0]),
                    p=self.jth_center_probabilities)
                # index = np.random.randint(0, np.shape(self.samples)[0]) # Generate
                    random index for cluster center
                if not any((self.samples[index] == cluster_mean).all() for cluster_mean in
                    self.cluster_means):
                    self.cluster_means.append(self.samples[index])
                    if k_meansplusplus and len(self.cluster_means) < self.num_clusters:
                        self.assign_data_points(update_jth_center_probability=True)
        else:
            for i in range(self.num_clusters):
                self.cluster_means[i] =
                    np.mean(self.samples[np.array(self.one_hot_assignments[:, i],
                    dtype=bool)], axis=0)

    def assign_data_points(self, update_jth_center_probability=False):

        distances = np.array([]) # Euclidean distance list of each sample to each cluster
            center

        for cluster_mean in self.cluster_means:
            distance_to_cluster = np.square(np.linalg.norm(self.samples - cluster_mean,
                axis=1))
            # distance_to_cluster = np.square(np.linalg.norm(self.samples - cluster_mean,
                axis=1))
```

```
            if np.any(distances): # If not empty
                distances = np.c_[distances, distance_to_cluster] # Append norm of kth
                    cluster to array
            else:
                distances = distance_to_cluster # If empty, create matrix with norm of
                    first cluster

        self.one_hot_assignments =
            np.eye(len(self.cluster_means))[np.argmin(distances.reshape(-1,
            len(self.cluster_means)), axis=1).reshape(-1)] # One hot assignment
        self.distortion =
            sum([np.sum(np.square(np.linalg.norm(self.samples[np.array(self.one_hot_assignments[:,i],
            dtype=bool)]- self.cluster_means[i], axis=1))) for i in
            range(len(self.cluster_means))])
        if update_jth_center_probability:
            self.jth_center_probabilities =
                np.divide(distances[np.array(self.one_hot_assignments,
                dtype=bool).squeeze()], self.distortion)
        else:
            self.num_iterations += 1

    def print_info(self):
        print("K = {} clusters\nTotal number of iterations: {}\nDistortion: {}\n\n".
             format(self.num_clusters, self.num_iterations, self.distortion))


class Expectation_Maximization():

    def __init__(self, kmean_estimation):
        self.num_clusters = kmean_estimation.num_clusters
        self.samples = kmean_estimation.samples
        self.mixture_means = kmean_estimation.cluster_means
        self.num_iterations = 0
        self.posterior_probabilities = kmean_estimation.one_hot_assignments
        self.kth_effective_points = np.sum(self.posterior_probabilities, axis=0)
        self.priors = self.kth_effective_points / len(self.samples)
        self.mixture_covariences = list(np.zeros([self.num_clusters,
            self.samples.shape[1], self.samples.shape[1]]))
        self.update_mixture_covariences()
        self.update_cost()

    def update_conditional_probabilities(self):
        self.cond_probabilites = np.zeros(self.posterior_probabilities.shape)
        self.log_cond_probabilites = np.zeros(self.posterior_probabilities.shape)
        for k in range(self.num_clusters):
            gaussian_distribution = multivariate_normal(self.mixture_means[k],
                self.mixture_covariences[k])
            self.cond_probabilites[:, k] = gaussian_distribution.pdf(self.samples)
            self.log_cond_probabilites[:, k] = gaussian_distribution.logpdf(self.samples)

    def update_posterior_probabilities(self):
        # self.update_conditional_probabilities()
        self.posterior_probabilities = (1/(np.matmul(self.cond_probabilites,
            self.priors)).reshape([-1, 1]))*(self.cond_probabilites*self.priors)
```

```
        self.kth_effective_points = np.sum(self.posterior_probabilities, axis=0)


    def update_priors(self):
        self.priors = self.kth_effective_points / len(self.samples)

    def update_mixture_means(self):
        for k in range(self.num_clusters):
            self.mixture_means[k] =
                np.sum(self.posterior_probabilities[:,k].reshape([-1,1])*self.samples,
                axis=0) / self.kth_effective_points[k]

    def update_mixture_covariences(self):
        for k in range(self.num_clusters):
            self.mixture_covariences[k] = np.sum(
                [self.posterior_probabilities[i,
                    k]*np.outer(self.samples[i]-self.mixture_means[k],
                    self.samples[i]-self.mixture_means[k])
                 for i in range(len(self.samples))], axis=0) / self.kth_effective_points[k]

    def update_cost(self):
        self.cost = 0
        self.update_conditional_probabilities()
        for k in range(self.num_clusters):
             for i in range(len(self.samples)):
                 self.cost += self.posterior_probabilities[i, k] *
                     (self.log_cond_probabilites[i, k] + math.log(self.priors[k]))
        self.num_iterations += 1


def find_quasi_orthogonal_basis(dim, num_basis, from_memory=False):
    if from_memory:
        # return sample_generation.generate_d_dim_samples(dim, num_basis,
            best_seeds=[3770, 350, 86, 861, 5616, 2738])
        return sample_generation.generate_d_dim_samples(dim, num_basis,
            best_seeds=np.load('best_seeds.npy'))
    else:
        cost_list = []
        best_seeds = [3770]
        for j in range(2, num_basis + 1):
            for i in range(10000):
                trial_seed = list(best_seeds)
                trial_seed.append(i)
                q_orthoganal_vectors = sample_generation.generate_d_dim_samples(dim, j,
                    best_seeds=trial_seed)
                normalized_vectors = q_orthoganal_vectors /
                    np.linalg.norm(q_orthoganal_vectors, axis=1).reshape([-1, 1])
                check_orthogonality = normalized_vectors.dot(normalized_vectors.T)
                cost = np.sum(abs(check_orthogonality -
                    np.diag(check_orthogonality.diagonal())))
                if np.count_nonzero(q_orthoganal_vectors[j-1]) in range((dim//3) -
                    (dim//3)//4, (dim//3) + (dim//3)//4):
                    cost_list.append(cost)
                else:
```

```
                cost_list.append(100)
            print("Best cost: {}\nBest seed: {}\n".format(min(cost_list),
                cost_list.index(min(cost_list))))
            best_seeds.append(cost_list.index(min(cost_list)))
            cost_list.clear()
        np.save('best_seeds.npy', best_seeds)
        return sample_generation.generate_d_dim_samples(dim, num_basis,
            best_seeds=best_seeds)


def generate_d_dim_data(basis, num_samples=50):
    #  Create component coefficients V1, V2, component choice, and Noise vectors
    np.random.seed(785) #785
    v1 = np.random.normal(size=num_samples)
    np.random.seed(3589) #3589
    v2 = np.random.normal(size=num_samples)
    random_state = np.random.RandomState(13489) # 13489
    noise = random_state.multivariate_normal(np.zeros(len(basis[0])),
        0.01*np.eye(len(basis[0])), num_samples)
    # np.random.seed(89) #89
    choice = np.random.choice([1,2,3], p=[1/3, 1/3, 1/3], size=num_samples)
    samples = np.zeros([num_samples, len(basis[0])])
    one_hot_labels = np.zeros([num_samples, 3])
    for i in range(num_samples):
        if choice[i] == 1:
            samples[i, :] = basis[0] + v1[i]*basis[1] + v2[i]*basis[2] + noise[i]
            one_hot_labels[i, :] = np.array([1, 0, 0])
        elif choice[i] == 2:
            samples[i, :] = 2*basis[3] + math.sqrt(2)*v1[i]*basis[4] + v2[i]*basis[5] +
                noise[i]
            one_hot_labels[i, :] = np.array([0, 1, 0])
        else:
            samples[i, :] = math.sqrt(2)*basis[5] + v1[i]*(basis[0] + basis[1]) +
                1/math.sqrt(2)*v2[i]*basis[4] + noise[i]
            one_hot_labels[i, :] = np.array([0, 0, 1])
    return samples, one_hot_labels

def generate_d_dim_data2(basis, num_samples=50):
    #  Create component coefficients V1, V2, component choice, and Noise vectors
    np.random.seed(785) #785
    v1 = np.random.normal(size=num_samples)
    np.random.seed(3589) #3589
    v2 = np.random.normal(size=num_samples)
    random_state = np.random.RandomState(13489) # 13489
    noise = random_state.multivariate_normal(np.zeros(len(basis[0])),
        0.01*np.eye(len(basis[0])), num_samples)
    # np.random.seed(89) #89
    choice = np.random.choice([+1, -1], p=[1/2, 1/2], size=num_samples)
    samples = np.zeros([num_samples, len(basis[0])])
    one_hot_labels = np.zeros([num_samples, 3])
    for i in range(num_samples):
        if choice[i] == 1:
            samples[i, :] = basis[0] + v1[i]*basis[1] + v2[i]*basis[2] + noise[i]
            one_hot_labels[i, :] = np.array([1, 0, 0])
```

```
        elif choice[i] == 2:
            samples[i, :] = 2*basis[3] + math.sqrt(2)*v1[i]*basis[4] + v2[i]*basis[5] +
                noise[i]
            one_hot_labels[i, :] = np.array([0, 1, 0])
        else:
            samples[i, :] = math.sqrt(2)*basis[5] + v1[i]*(basis[0] + basis[1]) +
                1/math.sqrt(2)*v2[i]*basis[4] + noise[i]
            one_hot_labels[i, :] = np.array([0, 0, 1])
    return samples, one_hot_labels


def run_Kmeans_algorithm(samples, one_hot_labels, rangeK=np.arange(1, 10),
    num_trials=100, k_means_plusplus=False, plot_tables=False):

    def plot_distortion():
        # Plot the distortion as a function of K.
        fig = plt.figure()
        distortions = [k_means.distortion for k_means in k_means_list]
        plt.plot(rangeK, distortions, 'ro')
        plt.title("Distortion as a function of K")
        plt.xlabel("Number of clusters")
        plt.ylabel("Distortion")

        # Plot the ratio as a function of K.
        fig2 = plt.figure()
        distortion_ratios = [k_means_list[i + 1].distortion / k_means_list[i].distortion
            for i in range(len(k_means_list) - 1)]
        x_labels = ["K={}/K={}".format(i+1, i) for i in rangeK[:len(rangeK)-1]]
        plt.plot(x_labels, distortion_ratios, 'bo')
        plt.title("Rate of distortion change")
        plt.xlabel("Ratio of K = i+1 / K = i")
        plt.ylabel("Distortion change")
        plt.show()

    # K-means Algorithm
    jth_k_mean = 0
    epsilon = 0.000001
    k_means_list = []
    probability_matrix = []
    for num_clusters in rangeK: # For K = 2, 3, 4, 5

        for trial in range(num_trials): # Get best estimate from num_trials

            k_means_trial = K_means(num_clusters, samples)
            k_means_trial.update_cluster_centers(initial_assignment=True,
                k_meansplusplus=k_means_plusplus) # Change for k-means++
            k_means_trial.assign_data_points()
            while k_means_trial.num_iterations < 20:
                previous_means = k_means_trial.cluster_means.copy()
                k_means_trial.update_cluster_centers()
                k_means_trial.assign_data_points()
                if np.all(abs(np.subtract(k_means_trial.cluster_means, previous_means)) <
                    epsilon):
                    break
```

```
            # k_means_trial.print_info()
            try:
                if k_means_trial.distortion < k_means_list[jth_k_mean].distortion: #
                    Append trial with least distortion
                    k_means_list[jth_k_mean] = k_means_trial
            except IndexError:
                k_means_list.append(k_means_trial)

        print("Best Iteration:")
        k_means_list[jth_k_mean].print_info()

        L = np.count_nonzero(one_hot_labels, axis=0) # Number of samples from each class L
        kth_probability_matrix = []
        for l in range(one_hot_labels.shape[1]): # For each mixture component
            ith_label_indices = np.nonzero(one_hot_labels[:, l])[0] # Get sample indices
                generate from lth component
            K = np.array(
                [np.count_nonzero(k_means_list[jth_k_mean].one_hot_assignments[ith_label_indices,
                    k], axis=0) for k in
                 range(num_clusters)]).squeeze() # Number of samples assigned to K that
                     came from L
            kth_probability_matrix.append(np.divide(K, L[l]))

        probability_matrix.append(np.array(kth_probability_matrix))
        jth_k_mean += 1

    # Plot empirical probability tables
    if plot_tables:
        plot_probability_tables(probability_matrix) # Generates plot figures
        plot_distortion() # Plot the distortion as a function of K.
    for i in range(len(probability_matrix)): print("{}K = {}\n{}\n".format((2 * i + 2) *
        " ", i + 2, probability_matrix[i]))

    return k_means_list


def run_EM_algorithm(kmeans_tests, one_hot_labels, plot_tables=False, epsilon=0.0001):
    jth_estimation = 0
    gaussian_mixture_list = []
    probability_matrix = []
    for kmeans_test in kmeans_tests:
        mixture_estimation = Expectation_Maximization(kmeans_test)
        while mixture_estimation.num_iterations < 300:
            mixture_estimation.update_posterior_probabilities()
            mixture_estimation.update_priors()
            mixture_estimation.update_mixture_means()
            mixture_estimation.update_mixture_covariences()
            previous_cost = mixture_estimation.cost
            mixture_estimation.update_cost()
            print("Iteration : {}\nCost: {}\n".format(mixture_estimation.num_iterations,
                mixture_estimation.cost))
            if abs(mixture_estimation.cost-previous_cost) < epsilon:
                break
        gaussian_mixture_list.append(mixture_estimation)
```

```
        # pprint.pprint(mixture_estimation.__dict__)
        pprint.pprint(mixture_estimation.num_iterations)
        pprint.pprint(mixture_estimation.cost)

        L = np.count_nonzero(one_hot_labels, axis=0) # Number of samples from each class L
        kth_probability_matrix = []
        for l in range(one_hot_labels.shape[1]):
            ith_label_indices = np.nonzero(one_hot_labels[:, l])[0] # Get sample indices
                generate from lth component
            K = np.array(
                [np.sum(gaussian_mixture_list[jth_estimation].posterior_probabilities[ith_label_indices,
                    k], axis=0) for k in
                 range(gaussian_mixture_list[jth_estimation].num_clusters)]).squeeze() #
                     Number of samples assigned to K that came from L

            kth_probability_matrix.append(np.divide(K, L[l]))

        probability_matrix.append(np.array(kth_probability_matrix))
        jth_estimation += 1

    # Plot empirical probability tables
    if plot_tables: plot_probability_tables(probability_matrix) # Generates plot figures
    for i in range(len(probability_matrix)): print("{}K = {}\n{}\n".format((2 * i + 2) *
        " ", i + 2, probability_matrix[i]))
    return gaussian_mixture_list


def generate_2D_test_data(num_samples=200, plot_data=False):
    seed1, seed2, seed3 = 14, 8, 1228 # Seed for each class 14, 8, 1228
    class_1, _, class_2, class_3 = sample_generation.generate_samples(num_samples,
        seed0=seed1, seed1A=seed2, seed1B=seed3)
    class_1.prior, class_2.prior, class_3.prior = 1/2, 1/6, 1/3

    np.random.seed(seed1 + seed2 + seed3)
    prior = np.random.rand(num_samples)
    samples = np.zeros([num_samples, 2])
    one_hot_labels = np.zeros([num_samples, 3])
    for i in range(num_samples):
        if prior[i] < class_1.prior:
            samples[i, :] = class_1.data[i, :] # Matrix with samples
            one_hot_labels[i, :] = np.array([1, 0, 0]) # One hot encoding matrix
        elif prior[i] < class_1.prior + class_2.prior:
            samples[i, :] = class_2.data[i, :] # Matrix with samples
            one_hot_labels[i, :] = np.array([0, 1, 0]) # One hot encoding matrix
        else:
            samples[i, :] = class_3.data[i, :] # Matrix with samples
            one_hot_labels[i, :] = np.array([0, 0, 1]) # One hot encoding matrix

    if plot_data:
        plt.figure(1)
        plt.plot(class_1.data[np.equal(class_1.data, samples)[:, 0], 0],
                 class_1.data[np.equal(class_1.data, samples)[:, 1], 1], 'b.')
        plt.plot(class_2.data[np.equal(class_2.data, samples)[:, 0], 0],
                 class_2.data[np.equal(class_2.data, samples)[:, 1], 1], 'r.')
```

```
        plt.plot(class_3.data[np.equal(class_3.data, samples)[:, 0], 0],
                 class_3.data[np.equal(class_3.data, samples)[:, 1], 1], 'g.')
        plt.axis('equal')
        plt.legend(['Component 1', 'Component 2', 'Component 3'])
        plt.title('Generated 2D samples')
        plt.xlabel('dim 1')
        plt.ylabel('dim 2')

        plt.figure(2)
        plt.plot(samples[:, 0], samples[:, 1], 'k.')
        plt.axis('equal')
        plt.legend(['Samples'])
        plt.title('Generated 2D samples')
        plt.xlabel('dim 1')
        plt.ylabel('dim 2')
        plt.show()
    return samples, one_hot_labels


def plot_probability_tables(probability_matrix):
    for i in range(len(probability_matrix)):
        # Prepare table
        columns = np.arange(1, probability_matrix[i].shape[1] + 1)
        rows = np.arange(1, probability_matrix[i].shape[0] + 1)
        cell_data = probability_matrix[i]
        normal = plt.Normalize(cell_data.min() - 0.3, cell_data.max() + 0.3) # 0.3 works
            well
        colors = plt.cm.coolwarm(normal(cell_data))

        F = plt.figure(figsize=(15, 8))
        ax = F.add_subplot(111, frameon=True, xticks=[], yticks=[])
        ax.axis('tight')
        ax.axis('off')
        plt.xlabel("K")
        plt.ylabel("l")
        the_table = ax.table(cellText=cell_data, colWidths=[0.2] * columns.size,
                             colLabels=columns, rowLabels=rows, loc='center',
                                 cellColours=colors)
    plt.show()


def get_geometric_insight(test, basis):
    normalized_basis = basis / np.linalg.norm(basis, axis=1).reshape([-1, 1])
    mean_correlation = []
    for i in range(len(test)):
        normalized_means = test[i].mixture_means / np.linalg.norm(test[i].mixture_means,
            axis=1).reshape([-1, 1])
        # normalized_means = np.array(test[i])
        mean_cross_correlation = normalized_basis.dot(normalized_means.T)
        # mean_cross_correlation = normalized_means.dot(normalized_basis.T)
        # mean_cross_correlation = np.matmul(normalized_basis, normalized_means)
        mean_correlation.append(mean_cross_correlation)
    plot_probability_tables(mean_correlation)
```

```
if __name__ == '__main__':

    # # Generate 2D test samples
    # samples, one_hot_labels = generate_2D_test_data(num_samples=250, plot_data=False)
    # # Run Kmeans algorithm
    # run_Kmeans_algorithm(samples, one_hot_labels, rangeK=np.arange(2, 10),
        plot_tables=True) # Can only plot for K > 1
    # # Run Kmeans++ algorithm
    # run_Kmeans_algorithm(samples, one_hot_labels, rangeK=np.arange(2, 6),
        k_means_plusplus=False, plot_tables=True)
    #
    # # Run Expectation Maximization algorithm on 2D test samples
    # kmeans_tests = run_Kmeans_algorithm(samples, one_hot_labels, rangeK=np.arange(2,
        6), k_means_plusplus=True, plot_tables=False)
    # run_EM_algorithm(kmeans_tests, one_hot_labels, plot_tables=True)

    # Generate quasi-orthogonal basis
    q_orthoganal_basis = find_quasi_orthogonal_basis(dim=100, num_basis=6,
        from_memory=True)
    # Generate D-dim samples from basis
    d_dim_samples, one_hot_labels = generate_d_dim_data(q_orthoganal_basis,
        num_samples=350)

    # Run K-Means algorithm with K_means++ initialization
    d_dim_kmeans_tests = run_Kmeans_algorithm(d_dim_samples, one_hot_labels,
        rangeK=np.arange(2, 6), k_means_plusplus=True, plot_tables=False)
    get_geometric_insight(d_dim_kmeans_tests, q_orthoganal_basis)

    # # Run EM algorithm to estimate gaussian mixture parameters
    # d_dim_EM_tests = run_EM_algorithm(d_dim_kmeans_tests, one_hot_labels,
        plot_tables=False, epsilon=0.001)
    # get_geometric_insight(d_dim_EM_tests, q_orthoganal_basis)


import unsupervised_learning as ul
import matplotlib.pyplot as plt
import sample_generation
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import multivariate_normal
from scipy.linalg import svd
from sklearn.decomposition import PCA
from sklearn.linear_model import Lasso
from sklearn.linear_model import Ridge


import pprint
import math


def plot_s_wrt_N(q_orthoganal_basis):
    s_list = []
    for i in range(2 * 100, 100 * 100, 1000):
```

```
        # Generate D-dim samples from basis
        d_dim_samples, one_hot_labels = ul.generate_d_dim_data(q_orthoganal_basis,
            num_samples=i)
        U, s, Vh = svd(d_dim_samples-d_dim_samples.mean(0))
        s_list.append(s)
    s_matrix = np.array(s_list)
    np.save('s_matrix', s_matrix)

    plt.figure()
    plt.matshow(s_matrix)
    plt.title('Singular Values')
    plt.ylabel('N starts at 200\n, ends at 10,000 \nin steps of 1,000 \nincreasing from
        top to bottom')
    plt.xlabel('Singular value')
    plt.show()


if __name__ == '__main__':

    # Generate quasi-orthogonal basis
    q_orthoganal_basis = ul.find_quasi_orthogonal_basis(dim=100, num_basis=6,
        from_memory=True)

    # plot_s_wrt_N(q_orthoganal_basis)


    d_dim_samples, one_hot_labels = ul.generate_d_dim_data(q_orthoganal_basis,
        num_samples=250)

    pca = PCA(n_components=6) # pick first 6
    pca.fit(d_dim_samples)

    U, S, VT = np.linalg.svd(d_dim_samples - d_dim_samples.mean(0))

    X_train_pca = pca.transform(d_dim_samples)

    # Run K-Means algorithm with K_means++ initialization
    d_dim_kmeans_tests = ul.run_Kmeans_algorithm(X_train_pca, one_hot_labels,
        rangeK=np.arange(2, 6),
                                            k_means_plusplus=True, plot_tables=False)
    ul.get_geometric_insight(d_dim_kmeans_tests, q_orthoganal_basis)

    loss_lst = []
    for m in range(1, 50):
        # Reconstruction with L1 (Lasso) penalization
        # the best value of alpha was determined using cross validation
        # with LassoCV
        i  = i/100
        X_projected = pca.inverse_transform(X_train_pca)
        loss = ((d_dim_samples - X_projected) ** 2).mean()
        loss_lst.append(loss)
```