CS-E3210- Machine Learning Basic Principles
Home Assignment 5 - "Clustering"

A. Moskalev and A. Jung

Your solutions to the following problems should be submitted as one single pdf which does not contain any personal information (student ID or name). The only rule for the layout of your submission is that for each problem there has to be exactly one separate page containing the answer to the problem. You are welcome to use the LaTeX-file underlying this pdf, available under `https://version.aalto.fi/gitlab/junga1/MLBP2017Public`, and fill in your solutions there.

Problem 1: Hard Clustering

Answer. We have implemented the k-means algorithm (cf. `https://version.aalto.fi/gitlab/junga1/MLBP2017Public/blob/master/Clustering/mlbp17_Clustering_v1.pdf`). Every time the algorithm was run, the cluster means are initialized by randomly selecting data points

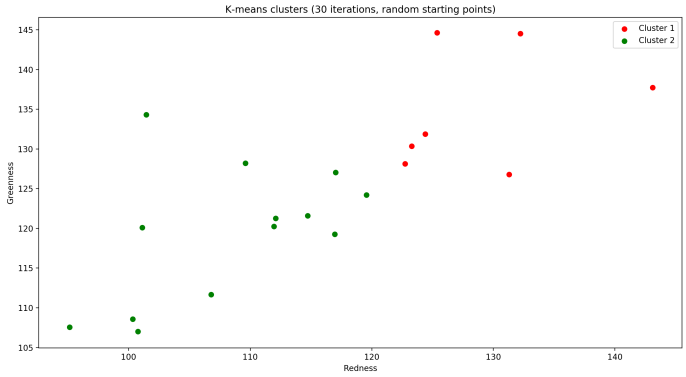

Figure 1: The cluster assignment delivered by k-means after 30 iterations

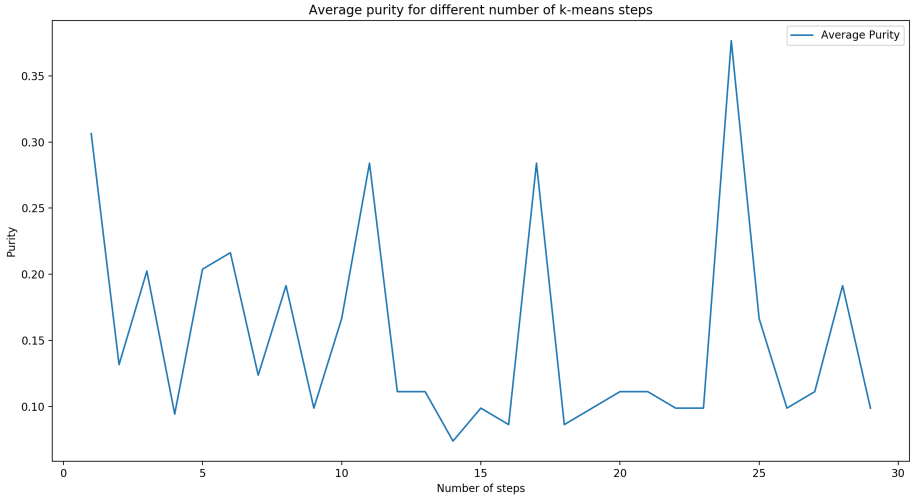

Figure 2: Average purity (using 10 independent runs of soft-clustering) as a function of iterations M used for k-means.

Problem 2: Soft Clustering

Answer. We perform soft clustering using GMM as discussed in `https://version.aalto.fi/gitlab/junga1/MLBP2017Public/blob/master/Clustering/mlbp17_Clustering_v1.pdf` using feature vectors $\mathbf{x}^{(i)} \in \mathbb{X}$ of randomly selected data points as initial cluster means and initial covariance matrices $\mathbf{C}_1 = \mathbf{C}_2 = \mathbf{I}$.

We found that it is beneficial to normalize the numerical values of the data points, in order to avoid dividing by small numbers in the soft-clustering calculations (e.g., in $\frac{\mathcal{N}(\mathbf{x}^{(i)}|\mathbf{m}_1,\mathbf{C}_1)}{\mathcal{N}(\mathbf{x}^{(i)}|\mathbf{m}_1,\mathbf{C}_1)+\mathcal{N}(\mathbf{x}^{(i)}|\mathbf{m}_2,\mathbf{C}_2)}$). In particular, we found that normalizing the feature vector $\mathbf{x}^{(i)}$ by 63.75 produces reasonable results.

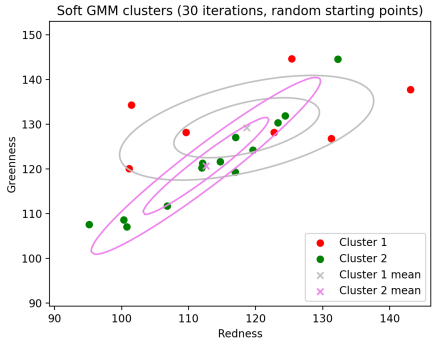

Figure 3: After the soft-clustering has been completed, we de-normalize the cluster means and covariances by multiplying with 63.75. We then plot the denormalized cluster means and covariance matrices along with the original feature vectors $\mathbf{x}^{(i)} \in \mathbb{X}$.

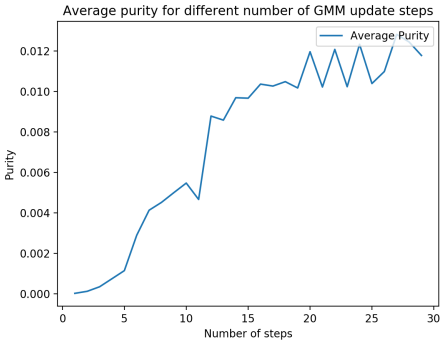

Figure 4: Average purity (using 10 independent runs of soft-clustering) as a function of iterations M used for soft-clustering algorithm.

Code for this problem is in Appendix B.

Appendix A

Python source code for Problem 1

```
from PIL import Image
import numpy as np
import glob
import os
from matplotlib import pyplot as pl

def main():
  dir_name = "/Users/artemmoskalev/Desktop/ML_clustering/*.jpeg"

  X, summer, winter = compute_features(dir_name)

  repeats = 10
  Mmax = 30
  purity_list = []
  for M in range(1, Mmax):
    purity_sum = 0.0
    for i in range(repeats):
      cluster1, cluster2 = k_means(X, M)
      purity_sum = purity_sum + get_purity(cluster2, summer, winter)
    purity_list.append(purity_sum/repeats)

  pl.scatter(X[cluster1, 0], X[cluster1, 1], c='red')
  pl.scatter(X[cluster2, 0], X[cluster2, 1], c='green')
  pl.legend(labels=["Cluster 1", "Cluster 2"], loc=1)
  pl.title("K-means clusters (30 iterations, random starting points)")
  pl.xlabel("Redness")
  pl.ylabel("Greenness")
  pl.show()

  # plot results: number of iterations on x-axis, loss function value on y-axis
  pl.plot([i for i in range(1, Mmax)], purity_list)
  pl.legend(labels=["Average Purity"], loc=1)
  pl.title("Average purity for different number of k-means steps")
  pl.xlabel("Number of steps")
  pl.ylabel("Purity")
  pl.show()

def k_means(X, M):
  point_1 = draw_random(X)
  point_2 = draw_random(X)
  for i in range(M):
    cluster1, cluster2 = find_cluster_points(X, point_1, point_2)
    point_1, point_2 = redefine_centers(X, cluster1, cluster2)
  return (cluster1, cluster2)

# takes 2 cluster centers, returns 2 clusters of points
def find_cluster_points(X, center1, center2):
  cluster1 = []
  cluster2 = []
  for i in range(X.shape[0]):
    clustered_point = X[i]
    distance1 = np.linalg.norm(clustered_point - center1)
```

```
    distance2 = np.linalg.norm(clustered_point - center2)
    if distance1 > distance2:
      cluster1.append(i)
    else:
      cluster2.append(i)
  return (cluster1, cluster2)

# redefine where the center of the cluster should be
def redefine_centers(X, cluster1, cluster2):
  cluster1_sum = np.sum(X[cluster1, :], axis=0)
  cluster2_sum = np.sum(X[cluster2, :], axis=0)
  cluster1_divisor = len(cluster1) if len(cluster1) != 0 else 1
  cluster2_divisor = len(cluster2) if len(cluster2) != 0 else 1
  return (cluster1_sum/cluster1_divisor, cluster2_sum/cluster2_divisor)

def draw_random(X):
  random_red, random_green = X[np.random.randint(X.shape[0]), :]
  return np.asarray([random_red, random_green])

# these 2 methods compute purity for the cluster 2 and 2 real label sets
def get_purity(cluster2, summer, winter):
  cluster2_set = set(cluster2)
  summer_set = set(summer)
  winter_set = set(winter)

  pw_arg = len(cluster2_set.intersection(winter_set))/len(winter_set)
  ps_arg = len(cluster2_set.intersection(summer_set))/len(summer_set)

  return 0.5*(compute_purity(pw_arg) + compute_purity(ps_arg))

def compute_purity(p):
  p_term = (p * np.log2(p)) if (p != 0) else 0
  p_1_term = ((1 - p) * np.log2(1 - p)) if (1 - p != 0) else 0
  return 1 + p_term + p_1_term

def compute_features(filedir):
  # iterate all images in a given folder
  matrix = np.zeros((len(glob.glob(filedir)), 2))
  winter = []
  summer = []
  index = 0
  for filename in glob.glob(filedir):
    basename = os.path.basename(filename)
    if basename.startswith('winter'):
      winter.append(index)
    else:
      summer.append(index)
    image = Image.open(filename)

    height = np.size(image, 0)
    width = np.size(image, 1)
    pixel_sum = np.sum(np.array(image), axis=(0, 1))
    redness = pixel_sum[0]/(width*height)
    greenness = pixel_sum[1]/(width*height)
    matrix[index, :] = [redness, greenness]
    index = index + 1
```

```
  return (matrix, summer, winter)

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

Appendix B

Python source code for Problem 2

```
from PIL import Image
import numpy as np
import glob
import os
from matplotlib import pyplot as pl
from scipy.stats import multivariate_normal

def main():
  dir_name = "/Users/artemmoskalev/Desktop/ML_clustering/*.jpeg"

  X, summer, winter = compute_features(dir_name)

  repeats = 10
  Mmax = 30
  purity_list = []
  for M in range(1, Mmax):
    purity_sum = 0.0
    for i in range(repeats):
      likelihoods, mean1, c1, mean2, c2 = k_means_soft(X, M)
      purity_sum = purity_sum + get_purity(likelihoods, summer, winter)
      cluster1, cluster2 = find_clusters(likelihoods)
    purity_list.append(purity_sum/repeats)

  xlist = np.linspace(1.4, 2.2, 100)
  ylist = np.linspace(1.4, 2.4, 100)
  x_grid1, y_grid1 = np.meshgrid(xlist, ylist)
  z_grid1 = pl.mlab.bivariate_normal(x_grid1, y_grid1, np.sqrt(c1[0, 0]), np.sqrt(c1[1, 1]), mean1[0], mean1[1], c1
  pl.contour(x_grid1 * 63.75, y_grid1 * 63.75, z_grid1, 2, colors='silver')

  x_grid2, y_grid2 = np.meshgrid(xlist, ylist)
  z_grid2 = pl.mlab.bivariate_normal(x_grid2, y_grid2, np.sqrt(c2[0, 0]), np.sqrt(c2[1, 1]), mean2[0], mean2[1], c2
  pl.contour(x_grid2 * 63.75, y_grid2 * 63.75, z_grid2, 2, colors='violet')

  mean1 = mean1 * 63.75
  mean2 = mean2 * 63.75
  print("MEAN 1: " + str(mean1) + ", MEAN 2: " + str(mean2))

  X = 63.75 * X
  pl.scatter(X[cluster1, 0], X[cluster1, 1], c='red')
  pl.scatter(X[cluster2, 0], X[cluster2, 1], c='green')
  pl.scatter([mean1[0]], [mean1[1]], c='silver', marker='x')
  pl.scatter([mean2[0]], [mean2[1]], c='violet', marker='x')
  pl.legend(labels=["Cluster 1", "Cluster 2", "Cluster 1 mean", "Cluster 2 mean"], loc=4)
  pl.title("Soft GMM clusters (30 iterations, random starting points)")
  pl.xlabel("Redness")
  pl.ylabel("Greenness")
  pl.show()

  # plot results: number of iterations on x-axis, loss function value on y-axis
  pl.plot([i for i in range(1, Mmax)], purity_list)
  pl.legend(labels=["Average Purity"], loc=1)
  pl.title("Average purity for different number of GMM update steps")
```

```
  pl.xlabel("Number of steps")
  pl.ylabel("Purity")
  pl.show()

def k_means_soft(X, M):
  mean1 = draw_random(X)
  c1 = np.identity(X.shape[1])
  mean2 = draw_random(X)
  c2 = np.identity(X.shape[1])
  for i in range(M + 1):
    likelihoods = find_degrees_of_belonging(X, mean1, c1, mean2, c2)
    mean1, c1, mean2, c2 = update_GMM_parameters(X, likelihoods)
  return (likelihoods, mean1, c1, mean2, c2)

# find clusters using likelihoods
def find_clusters(likelihoods):
  cluster1 = []
  cluster2 = []
  for i in range(len(likelihoods)):
    if likelihoods[i] >= 0.5:
      cluster1.append(i)
    else:
      cluster2.append(i)
  return (cluster1, cluster2)

# takes 2 cluster centers, returns likelihoods of belonging to cluster 1
def find_degrees_of_belonging(X, mean1, c1, mean2, c2):
  degrees = np.zeros(X.shape[0])
  for i in range(X.shape[0]):
    clustered_point = X[i]
    denominator = (multivariate_normal.pdf(clustered_point, mean1, c1) +
                   multivariate_normal.pdf(clustered_point, mean2, c2))
    degrees[i] = multivariate_normal.pdf(clustered_point, mean1, c1)/denominator
  return degrees

# redefine where the center of the cluster should be
def update_GMM_parameters(X, likelihoods):
  n1 = np.sum(likelihoods)
  n2 = likelihoods.shape[0] - n1

  # updating the distribution of cluster 1
  running_mean_sum = np.zeros(2)
  for i in range(X.shape[0]):
    running_mean_sum = running_mean_sum + X[i] * likelihoods[i]
  mean1_new = running_mean_sum/n1

  running_covariance_sum = np.zeros((2, 2))
  for i in range(X.shape[0]):
    running_covariance_sum = running_covariance_sum + np.outer(X[i] - mean1_new, X[i] - mean1_new) * likelihoods[i]
  c1_new = running_covariance_sum/n1

  # updating the distribution of cluster 2
  running_mean_sum = np.zeros(2)
  for i in range(X.shape[0]):
    running_mean_sum = running_mean_sum + X[i] * (1 - likelihoods[i])
  mean2_new = running_mean_sum/n2
```

```
  running_covariance_sum = np.zeros((2, 2))
  for i in range(X.shape[0]):
    running_covariance_sum = running_covariance_sum + np.outer(X[i] - mean1_new, X[i] - mean1_new) * (1 - likelihoo
  c2_new = running_covariance_sum/n2

  return (mean1_new, c1_new, mean2_new, c2_new)

# create random 2-d points (red, green) withing range of data
def draw_random(X):
  random_red, random_green = X[np.random.randint(X.shape[0]), :]
  return np.asarray([random_red, random_green])

# these 2 methods compute purity for the cluster 2 and 2 real label sets
def get_purity(likelihoods, summer, winter):
  pw_arg = 2*np.sum(likelihoods[winter])/(len(winter) + len(summer))
  ps_arg = 2*np.sum(likelihoods[summer])/(len(winter) + len(summer))

  return 0.5*(compute_purity(pw_arg) + compute_purity(ps_arg))

def compute_purity(p):
  p_term = (p * np.log2(p)) if (p != 0) else 0
  p_1_term = ((1 - p) * np.log2(1 - p)) if (1 - p != 0) else 0
  return 1 + p_term + p_1_term

# when computing features, normalize by 63.75, setting effective variable range max = 4 (2 std of initial normal)
def compute_features(filedir):
  # iterate all images in a given folder
  matrix = np.zeros((len(glob.glob(filedir)), 2))
  winter = []
  summer = []
  index = 0
  for filename in glob.glob(filedir):
    basename = os.path.basename(filename)
    if basename.startswith('winter'):
      winter.append(index)
    else:
      summer.append(index)
    image = Image.open(filename)

    height = np.size(image, 0)
    width = np.size(image, 1)
    pixel_sum = np.sum(np.array(image), axis=(0, 1))
    redness = pixel_sum[0]/(width*height)
    greenness = pixel_sum[1]/(width*height)
    matrix[index, :] = [redness/63.75, greenness/63.75]
    index = index + 1
  return (matrix, summer, winter)

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