Αναφορά 2ης σειράς ασκήσεων:

“Μείωση διάστασης και Ομαδοποίηση εικόνων”

Μάθημα: “ΜΥΕ002 - Μηχανική Μάθηση”

Ακαδημαϊκό έτος: 2021-2022

Πανεπιστήμιο Ιωαννίνων

Τμήμα Μηχανικών Η/Υ και Πληροφορικής

Ομάδα 76:

Μπουλώτης Παναγιώτης ΑΜ: 4271

Ζέζος Αργύριος ΑΜ: 4588

Εισαγωγή

Για τις ανάγκες αυτού του σετ ασκήσεων, χρησιμοποιήθηκε το notebook που παρέχει η ιστοσελίδα Kaggle, καθώς προσφέρει την άνεση να υλοποιήσουμε απευθείας τα δεδομένα του διαγωνισμού της. Οπότε στην συνέχεια θα ακολουθήσει εξήγηση κάθε κελιού (cell). Συνιστάται να εκτελεστούν τα κελιά με την σειρά, καθώς το κάθε κελί περιέχει πληροφορία που θα χρειαστούν τα επόμενα. Για να τρέξουμε το notebook αυτό πρέπει να πάμε στον διαγωνισμό της άσκησης, να πατήσουμε new notebook στην επιλογή code και να κάνουμε import τον κώδικα.

Διαχείριση δεδομένων

Στην αρχή κάνουμε εισαγωγή ορισμένες βασικές βιβλιοθήκες που θα χρειαστούμε στο project μας. Καθώς τα δεδομένα τα λαμβάνουμε από το Kaggle δημιουργήθηκε το path για ευκολία στην συνέχεια. Οπότε το πρώτο κελί:

```
#import basic libraries and setup dataset path

import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv) # image pr
ocessing
#from PIL import Image # image processing
#import lodgepole.image_tools as lit # linear approximation of gamma correction K
AGGLE WILL NOT IMPORT THIS
import os

path = '/kaggle/input' #the path of the directory that the dataset folder lies, c
hange to run on a different machine
```

Ακολούθως, πρέπει από τα δεδομένα του train_data του διαγωνισμού να ξεχωρίσουμε τα άτομα που θέλουμε, δηλαδή από τα 4000 συνολικά, να επιλέξουμε τυχαία 10 άτομα που έχουν τουλάχιστον 50 εικόνες (file_count ≥ 50) και να τα αποθηκεύσουμε στο πίνακα people. Άρα το δεύτερο κελί:

```
#randomly pick people of which there are least 50 pictures

import random

people = [] # list of randomly selected people of which there are
                                                                    least 50 pictures
while len(people)<10:
    r = random.randint(1, 4000)
    _, _, files = next(os.walk(path+'/11785-spring2021-hw2p2s1-face-classificatio
n/train_data/'+str(r)))
    file_count = len(files)
    if file_count>=50:
        people.append(r)
```

Στο τρίτο κελί, επιλέγουμε σε κάθε άτομο τις 50 πρώτες εικόνες του, τις μετατρέπουμε σε μονοχρωματικές με την χρήση της color.rbg2gray() και τις αποθηκεύουμε στο training πίνακα, ο οποίος θα περιέχει γενικά τα δεδομένα που θα αξιοποιήσουμε στη συνέχεια.

```
#get 50 images of each person picked and put them in the training array

from skimage import color
from skimage import io
# color_img = np.asarray(Image.open(img_filename)) / 255
# gray_img = lit.rgb2gray_approx(color_img)
#KAGGLE WONT IMPORT lit, skimage used instead below

training = list() #contains all the images to be used (500). 50 continous of each person and in the order the people were picked
for p in people:
    c=0
    for dirname, _, filenames in os.walk(path+'/11785-spring2021-hw2p2s1-face-classification/train_data/'+str(p)):
        for filename in filenames:
            if c<50:
                img = color.rgb2gray(io.imread(path+'/11785-spring2021-hw2p2s1-face-classification/train_data/'+str(p)+'/'+filename))
                training.append(img.flatten())#images need to be flat (vectors and not arrays)
                c+=1
```

Στη συνέχεια δημιουργήσαμε τον πίνακα y_true όπου έχει στη σειρά κάθε 50 θέσεις το αναγνωριστικό του ατόμου που ανταποκρίνεται η φωτογραφία, εφόσον τοποθετούνται στο training με την σειρά, δηλαδή από το πρώτο άτομο (άτομο 0) εώς το δέκατο (άτομο 9). Το τέταρτο κελί:

```
y_true = list() #indicates how training array was actually made, used for evaluation
for i in range(0,10):
    for j in range(0,50):
        y_true.append(i)
```

Μείωση διάστασης με PCA

Έχοντας έτοιμα τα δεδομένα, με την χρήση της **PCA()** από τη βιβλιοθήκη sklearn, μπορούμε να μειώσουμε την διάσταση των δεδομένων σε 25, 50 και 100 διαστάσεις, αποθηκεύοντας τα στην λίστα pca_reduced στις θέσεις pca_reduced[0], pca_reduced[1] και pca_reduced[2] αντίστοιχα.

Αυτήν την λίστα θα την χρησιμοποιήσουμε στην συνέχεια όταν θέλουμε να κάνουμε ομαδοποίηση. Αντίστοιχα, το πέμπτο κελί:

```
#PCA implementation

from sklearn.decomposition import PCA

pca_reduced=list()

for j in [25, 50, 100]:
    pca = PCA(n_components=j)
    pca_reduced.append(pca.fit_transform(training))
```

Μείωση διάστασης με Autoencoder

Ομοίως με πριν, έχουμε αντίστοιχα την λίστα ac_reduced στην οποία θα αποθηκεύσουμε τα αποτελέσματα της μεθόδου. Για την πραγματοποίηση του στόχου αυτού θα υλοποιήσουμε το αντίστοιχο νευρωνικό δίκτυο autoencoder με την χρήση του keras της βιβλιοθήκης Tensorflow. Η βιβλιοθήκη αυτή προσφέρει πολύ χρήσιμα εργαλεία για την καλύτερη αναπαράσταση του δικτύου μέσα στον κώδικα.

Εφόσον ακολουθούμε την αρχιτεκτονική d-d/4-M-d/4-d, αυτό παρακάτω υλοποιείται ως εξής:

Αρχικά εφόσον μετατρέψουμε την training σε πίνακα που επεξεργάζεται το numpy, βρίσκουμε την τιμή του d και την αποθηκεύουμε στην ncol (ως αριθμό στηλών) μέσω της sX.shape[1] όπου φυσικά γνωρίζουμε από την εκφώνηση πως στην περίπτωση μας είναι 4096 (64x64). Αυτός επομένως θα είναι ο αριθμός των διαστάσεων στην είσοδο του δικτύου.

```
#autoencoder implementation

from keras.layers import Input, Dense
from keras.models import Model
from sklearn.model_selection import train_test_split
from numpy.random import seed

ac_reduced = list()
```

```
for i in [25, 50, 100]:
    sX = np.asarray(training)
    ncol = sX.shape[1]
    X_train=X_test=sX
    input_dim = Input(shape = (ncol, ))
```

Στη συνέχεια εφόσον εκτελούμε σε επανάληψη για όλες τις τιμές του M, αποθηκεύουμε την τωρινή στο encoding_dim. Οπότε πλέον προσθέτουμε το επόμενο νευρώνα που έχει συνάρτηση ενεργοποίησης την relu με διαστάσεις d/4 όπως και οι επόμενοι δύο με διαστάσεις M και d/4. Η έξοδος έχει τον ίδιο αριθμό διάστασης με την είσοδο, αλλά με συνάρτηση ενεργοποίησης την σιγμοειδή. Τα πρώτα 2 επίπεδα συμπίεσης ανταποκρίνονται στον κωδικοποιητή που συμπιέζει τα δεδομένα ενώ τα επόμενα 2 στον αποκωδικοποιητή.

Επομένως εφόσον γίνεται η διαμόρφωση του δικτύου, μπορεί να γίνει η εκπαίδευση του autoencoder φτιάχνοντας το αντίστοιχο μοντέλο. Στη συνέχεια εκπαιδεύουμε το μοντέλο του με τα X_train ώστε να εξάγουμε την πληροφορία που χρειαζόμαστε (δηλαδή τα X_test) και μέσω του μοντέλου encoder να κάνουμε πρόβλεψη άρα και να παράγουμε τα δεδομένα που θα αποθηκεύσουμε στην ac_reduced.

```
# DEFINE THE DIMENSION OF ENCODER ASSUMED i
    encoding_dim = i
    # DEFINE THE ENCODER LAYERS
    encoded1 = Dense(4096/4, activation = 'relu')(input_dim)
    encoded2 = Dense(encoding_dim, activation = 'relu')(encoded1)
    # DEFINE THE DECODER LAYERS
    decoded1 = Dense(4096/4, activation = 'relu')(encoded2)
    decoded2 = Dense(ncol, activation = 'sigmoid')(decoded1)
    # COMBINE ENCODER AND DECODER INTO AN AUTOENCODER MODEL
    autoencoder = Model(inputs = input_dim, outputs = decoded2)
    # CONFIGURE AND TRAIN THE AUTOENCODER
    autoencoder.compile(optimizer = 'adadelta', loss = 'binary_crossentropy')


    # Train Auto Encoder
    autoencoder.fit(X_train, X_train, epochs = 10, batch_size = 10, shuffle = Tru
e, validation_data = (X_test, X_test))
    # Use Encoder level to reduce dimension of train and test data
    encoder = Model(inputs = input_dim, outputs = encoded2)
    # Predict the new data using Encoder
    encoded_out = encoder.predict(X_test)

    ac_reduced.append(encoded_out)
```

Ομαδοποίηση με Agglomerative hierarchical clustering

Εφόσον έχουμε υλοποιήσει τις μεθόδους ομαδοποίησης πρέπει να εξετάσουμε την αποτελεσματικότητα τους στα δεδομένα που μας δίνεται και αυτό γίνεται με βάση τις μετρικές purity και f1 score στο έβδομο κελί. Αυτά υπολογίζονται με την χρήση του metrics της βιβλιοθήκης sklearn όπου για την purity χρησιμοποιείται ο πίνακας σύγχυσης.

```
#purity function implementation

from sklearn import metrics

def purity_score(y_true, y_pred):
    # compute contingency matrix (also called confusion matrix)
    contingency_matrix = metrics.cluster.contingency_matrix(y_true, y_pred)
    # return purity
    return np.sum(np.amax(contingency_matrix, axis=0)) / np.sum(contingency_matri
x)

#f1 function implementation

def f1_measure(y_true , y_pred):
    return metrics.f1_score(y_true, y_pred, average='micro')
```

Οπότε πλέον με την χρήση του πακέτου cluster της βιβλιοθήκης sklearn μπορούμε να υλοποιήσουμε την μέθοδο στο όγδοο κελί:

```
#Agglomerative Clustering implementation

from sklearn.cluster import AgglomerativeClustering

AC = AgglomerativeClustering(n_clusters = 10, affinity = 'euclidean', linkage = '
ward')
```

Και για το ελέγξουμε για τα δεδομένα από το PCA και Autoencoder:

```
print('Agglomerative Hierarchical Clustering')


print('PCA')
for i,j in zip(pca_reduced, [25,50,100]):
    labels = AC.fit_predict(i)
    purity = purity_score(y_true,labels)
    f1 = metrics.f1_score(y_true, labels, average='micro')
    print('M='+str(j)+' purity='+str(purity)+' f1='+str(f1))
print('Autoencoder')
for i,j in zip(ac_reduced, [25,50,100]):
    labels = AC.fit_predict(i)
    purity = purity_score(y_true,labels)
    f1 = f1_measure(y_true, labels)
    print('M='+str(j)+' purity='+str(purity)+' f1='+str(f1))
```

Παρακάτω φαίνεται το αποτέλεσμα μιας εκτέλεσης:

```
Agglomerative Hierarchical Clustering
PCA
M=25 purity=0.282 f1=0.108
M=50 purity=0.286 f1=0.122
M=100 purity=0.282 f1=0.104
Autoencoder
M=25 purity=0.196 f1=0.114
M=50 purity=0.264 f1=0.076
M=100 purity=0.24 f1=0.082
```

Ομαδοποίηση με K-means

Για τον αλγόριθμο k-means υπήρχαν έτοιμες υλοποιήσεις από βιβλιοθήκες όπως η sklearn, αλλά επειδή χρησιμοποιούσαν μόνο την Ευκλείδια απόσταση και δεν δεχόντουσαν την συνημιτονοειδή, προσπαθήσαμε να υλοποιήσουμε εμείς τον αλγόριθμο στο τελευταίο κελί της εργασίας.

Αρχικά ορίζουμε τις συναρτήσεις **euclidean_dist()**, με την χρήση της βιβλιοθήκης numpy και **cosine_dist()**, με την χρήση της βιβλιοθήκης spatial της scipy για να τις αξιοποιήσουμε στη συνέχεια στον αλγόριθμο, όπου χρειάζεται υπολογισμό απόστασης.

```
from scipy import spatial
import math
K=10 # number of classes


def euclidean_dist(value1, value2):
    return np.linalg.norm(value1 - value2)


def cosine_dist(value1, value2):
    return spatial.distance.cosine(value1, value2)
```

Ο k-means ορίζεται στην συνάρτηση K_means() που παίρνει ως ορίσματα το σύνολο δεδομένων X, την συνάρτηση απόστασης που θα χρησιμοποιήσουμε (“euclidean” για την ευκλείδια, αλλιώς συνημιτονοειδή) και το μέγιστο πλήθος επαναλήψεων (max_iterations).

Δημιουργούμε έναν πίνακα με κενούς προς το παρόν πίνακες, τον clusters (μεγέθους k, όσο και οι ομάδες) στον οποίο στη συνέχεια θα τοποθετήσουμε τις ετικέτες των δεδομένων κάθε ομάδας και στον πίνακα centroids θα αποθηκεύουμε τα κέντρα των ομάδων αυτών.

```
def K_means(X, function, max_iterations):
    clusters = [[] for _ in range(K)] # we need as many clusters as the classes
    centroids = []
```

Ως αρχικοποίηση των κέντρων επιλέγουμε k τυχαία x_i από τα δεδομένα και τα προσθέτουμε στον πίνακα centroids.

```
    # first we need to initialize the centroids randomly
    for j in range(K):
        index = random.randint(0,len(X)-1)
        centroids.append(X[index])
```

Στη συνέχεια αρχίζουμε την επαναληπτική διαδικασία. Σε κάθε επανάληψη για κάθε x_i των δεδομένων υπολογίζουμε την απόσταση του με το κέντρο μ_j κάθε ομάδας και τον υπολογίζουμε στον τοπικό πίνακα distances.

Στη συνέχεια στην τοπική μεταβλητή q αποθηκεύεται η ετικέτα ομάδας με την μικρότερη απόσταση, δηλαδή αποθηκεύουμε σε αντίστοιχη ομάδα την ετικέτα του στοιχείο x_i που είναι πιο κοντά στο μέσο της ομάδας.

Οπότε με αυτό τον τρόπο οι πίνακες του clusters γεμίζουν με ετικέτες των δεδομένων σε κάθε ομάδα.

```
    # For each iteration
    iteration = 0
    while(iteration<max_iterations):
        #print("Iteration " + str(iteration))
        # Creating clusters
        # for each x value we need to compute the distance with each centroid
        for x_id, x in enumerate(X):
            if(function == "euclidean"):
                distances = [euclidean_dist(x, centroids[j]) for j in range(K)]
            else: # cosine distance
                distances = [cosine_dist(x, centroids[j]) for j in range(K)]
            q = np.argmin(distances)
            clusters[q].append(x_id)
```

Τώρα ως 2ο βήμα του αλγορίθμου, πρέπει να επαναυπολογίσουμε τα “νέα” κέντρα κάθε ομάδας. Αυτό γίνεται αποθηκεύοντας την μέση τιμή των στοιχείων κάθε cluster στο αντίστοιχο μέσο.

Επιπλέον πρέπει να αποθηκεύσουμε τα κέντρα πριν αλλάξουν στο πίνακα prev_centroids, ώστε στη συνέχεια να ελέγξουμε την διαφορά τους στον πίνακα differences.

```
        # Now we have our clusters ready and we need to compute the new centroids
        prev_centroids = centroids
        centroids = [np.mean(cluster, axis = 0) for cluster in clusters]
```

Αν το άθροισμα των διαφορών τους είναι ίσο με μηδέν, τότε αυτή είναι συνθήκη να σταματήσουμε την επαναληπτική διαδικασία, οπότε κάνουμε break.

Αλλιώς αυξάνουμε τον αριθμό επανάληψης για να βρούμε τα νέα clusters.

```
        # We check if the centroids are not altered
        if(function == "euclidean"):
            differences = [euclidean_dist(prev_centroids[j], centroids[j]) for j
                                                                    in range(K)]
        else:
            differences = [cosine_dist(prev_centroids[j], centroids[j]) for j in
                                                                  range(K)]

        sum_dif = sum(differences)
        #print("Sum of differences: " + str(sum_dif))
        if sum_dif == 0:
            break

        iteration = iteration + 1
```

Εφόσον έχει τελειώσει η επαναληπτική διαδικασία, μπορούμε να υπολογίσουμε πλέον τον πίνακα y_pred, κοιτώντας σε κάθε cluster τα στοιχεία σε ποια από τις k ομάδες αντιστοιχεί.

Τελικά το επιστρέφουμε ως αποτέλεσμα του αλγορίθμου.

```
    # Now that clusters and centroids are created we will create y_pred
    y_pred = [[] for _ in range(len(X))]
    for j, cluster in enumerate(clusters):
        for i in cluster:
            y_pred[i] = j
    return y_pred
```

Στη συνέχεια εκτελούμε τον αλγόριθμο με τις διαφορετικές εκδοχές των ζητούμενων για να εξάγουμε τα αποτελέσματα που θέλουμε:

```
print("PCA \n")
# K_means with PCA
print("Euclidean distance")
for i,j in zip(pca_reduced, [25,50,100]):
    y_pred = K_means(i,"euclidean",100)
    purity = purity_score(y_true,y_pred)
    f1 = metrics.f1_score(y_true, y_pred, average='micro')
    print('M='+str(j)+' purity='+str(purity)+' f1='+str(f1))

print("\nCosine distance")
for i,j in zip(pca_reduced, [25,50,100]):
    y_pred = K_means(i,"cosine",100)
    purity = purity_score(y_true,y_pred)
    f1 = metrics.f1_score(y_true, y_pred, average='micro')
    print('M='+str(j)+' purity='+str(purity)+' f1='+str(f1))
```

```
print("\n-----------------------------------------------------------------")

# K_means with Autoencoder
# Eucleidan distance
print("Autoencoder \n")
print("Euclidean distance")
for i,j in zip(pca_reduced, [25,50,100]):
    y_pred = K_means(i,"euclidean",100)
    purity = purity_score(y_true,y_pred)
    f1 = metrics.f1_score(y_true, y_pred, average='micro')
    print('M='+str(j)+' purity='+str(purity)+' f1='+str(f1))

print("\nCosine distance")
for i,j in zip(pca_reduced, [25,50,100]):
    y_pred = K_means(i,"cosine",100)
    purity = purity_score(y_true,y_pred)
    f1 = metrics.f1_score(y_true, y_pred, average='micro')
    print('M='+str(j)+' purity='+str(purity)+' f1='+str(f1))
```

Σε κάθε εκτέλεση του αλγορίθμου τα αποτελέσματα θα είναι διαφορετικά, καθώς τα μέσα των ομάδων αρχικοποιούνται τυχαία.

Κατά τις δοκιμές εκτέλεσης όταν χρησιμοποιούσαμε την συνημιτονοειδής συνάρτηση ως μέθοδο για την απόσταση, είχαμε καλύτερα αποτελέσματα. Για παράδειγμα μερικές φορές ενώ δίναμε 100 ως μέγιστες επαναλήψεις, ο αλγόριθμος με cosine απόσταση μπορεί να τερμάτιζε και στις 12 επαναλήψεις, λόγω της αμεταβλητότητας των μέσων, ενώ αυτό δεν συνέβαινε στην περίπτωση της Ευκλείδιας. Παρακάτω φαίνεται το αποτέλεσμα μιας εκτέλεσης:

```
PCA

Euclidean distance
M=25 purity=0.162 f1=0.066
M=50 purity=0.1 f1=0.10000000000000002
M=100 purity=0.1 f1=0.10000000000000002

Cosine distance
M=25 purity=0.23 f1=0.082
M=50 purity=0.232 f1=0.102
M=100 purity=0.22 f1=0.11200000000000002

-----------------------------------------------------------------
Autoencoder

Euclidean distance
M=25 purity=0.198 f1=0.14
M=50 purity=0.104 f1=0.10000000000000002
M=100 purity=0.106 f1=0.10000000000000002

Cosine distance
M=25 purity=0.226 f1=0.114
M=50 purity=0.218 f1=0.092
M=100 purity=0.204 f1=0.11
```

Παρακάτω παρουσιάζονται τα συνολικά αποτελέσματα εκτέλεσης των μεθόδων:

Δεδομένα με την μέθοδο PCA:

Method	**Dimension of data(M)**	**Purity**	**F-measure**
K-means) (Euclidean distance)	25	0.162	0.066
	50	0.1	≅0.1
	100	0.1	≅0.1
K-means) (Cosine distance)	25	0.23	0.082
	50	0.232	0.102
	100	0.22	≅0.112
Agglomerative Hierarchical Clustering	25	0.282	0.108
	50	**0.286**	**0.122**
	100	0.282	0.104

Δεδομένα με την μέθοδο Autoencoder:

Method	**Dimension of data(M)**	**Purity**	**F-measure**
K-means) (Euclidean distance)	25	0.198	0.14
	50	0.104	≅0.1
	100	0.106	≅0.1
K-means) (Cosine distance)	25	0.226	0.114
	50	0.218	0.092
	100	0.204	0.11
Agglomerative Hierarchical Clustering	25	0.196	0.114
	50	0.264	0.076
	100	0.24	0.082

Οπότε παρατηρούμε πως την καλύτερη purity και f-measure ακρίβεια την πετύχαμε στην περίπτωση που η διάσταση ήταν M=50 από την μέθοδο PCA και χρησιμοποιούσαμε την Agglomerative Hierarchical ως μέθοδο ομαδοποίησης.

Σε όλες τις περιπτώσεις, αποδεικνύεται και μέσω των μετρικών, πως για τα συγκεκριμένα δεδομένα, η χρήση της cosine απόστασης είναι προτιμότερη.