עבודה 2: מדעי נתונים- אורן יקואל וגל אלון

1) a. $x_1, x_2, \dots, x_n$ משתנים מקריים ב"ת ומפולגים באותה צורה

x_i מתאר את מספר הניסיונות הדרושים עד להצלחה

בפעם ה-i. ידוע כי הניסוי מתפלג גיאומטרית, לכן:

$$P(x_i, \theta) = (1-\theta)^{x_i - 1} \cdot \theta$$

אז לכל עבור $1 \le i \le n$:

$$P(x_1, x_2, \dots, x_n, \theta) = (1-\theta)^{x_1-1} \cdot \theta \cdot \dots \cdot (1-\theta)^{x_n-1} \cdot \theta = \theta^n \cdot (1-\theta)^{(\sum_{i=1}^{n} x_i) - n}$$

$$\log(P(x_1, \dots, x_n, \theta)) = \log(\theta^n \cdot (1-\theta)^{(\sum_{i=1}^{n} x_i) - n})$$ \ נפעיל log על המשוואות

$$= \log\theta^n + \log((1-\theta)^{\sum_{i=1}^{n} x_i - n}) = n\log\theta + ((\sum_{i=1}^{n} x_i) - n) \cdot \log(1-\theta)$$

נסמן:

$$f(\theta) = n\log\theta + ((\sum_{i=1}^{n} x_i) - n) \cdot \log(1-\theta)$$

נמצא מקס':

$$f'(\theta) = \frac{n}{\theta} - \frac{(\sum_{i=1}^{n} x_i) - n}{1-\theta} = 0$$

$$\frac{n - \theta n - (\theta \sum_{i=1}^{n} x_i - \theta n)}{\theta - \theta^2} = 0$$

$$n - \theta n = \theta \sum_{i=1}^{n} x_i - \theta n$$

$$\frac{n}{\theta} = \sum_{i=1}^{n} x_i$$

$$\theta_{max} = \frac{n}{\sum_{i=1}^{n} x_i}$$

1. b. $x_1 = 2, x_2 = 3, x_3 = 3, x_4 = 3, x_5 = 6, x_6 = 2$ $n = 6$

נציב בנוסחה שקיבלנו בסעיף א':

$$\theta = \frac{6}{19} = 0.315$$

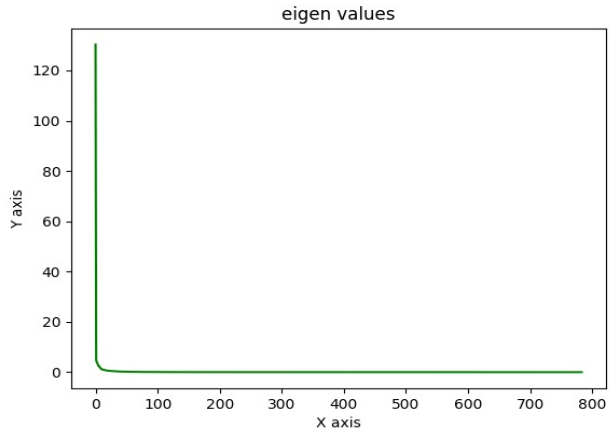
חלק 2- תיוג תמונות

<u>נתאר את האלגוריתם במילים:</u>

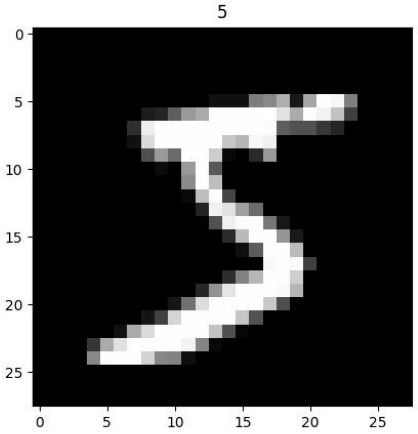
<u>א'+ב':</u>

גרף המאתר את הערכים הסינגולריים
ציר ה-X: ככמות הערכים הסינגולריים

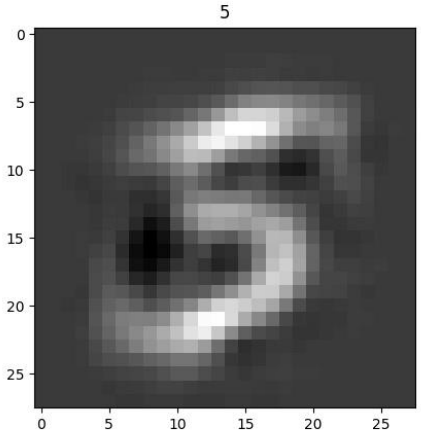
ציר ה-Y: ערך הערכים הסינגולריים



תמונה של הספרה המקורית - 5



תמונה של הספרה המשוחזרת - 5



ג'-ו':

הגענו ל49.6% הצלחה

ז':

הרצנו שוב את המודל 3 פעמים, התוצאות היו כדלהלן:

1. 47.19 אחוזי הצלחה
2. 43.5 אחוזי הצלחה
3. 58.8 אחוזי הצלחה

ח':

כאשר השתמשנו במימד קטן יותר (12) התוצאות היו: 45 אחוזי הצלחה.

הורדנו את המימד, ולכן ישנו פחות מידע שמתאר את התמונות, ולכן נצפה לאחוזי הצלחה נמוכים מאשר בסעיפים קודמים.

ט':

כאשר השתמשנו במרכזים שאנו הרכבנו (ולא רנדומליים כמו בסעיפים קודמים) התוצאות היו: 62 אחוזי הצלחה.

אכן ציפינו לקבל אחוזי הצלחה גבוהים יותר.

להלן הקוד:

בנוסף צירפנו לינק לGITHUB של גל- ובתוכו קבצי הקוד המלאים

https://github.com/GalAlon9/DS221_HW2

Main

```
import matplotlib.pyplot as plt
import numpy as np

import MnistDataloader
import DigitClassifying

loader = MnistDataloader.MnistDataloader('train-images.idx3-ubyte', 'train-
labels.idx1-ubyte'
                                         , 't10k-images.idx3-ubyte', 't10k-
labels.idx1-ubyte')
(x_train, y_train), (x_test, y_test) = loader.load_data()

# divide the images by 250 and decrease by 0.5
x_train = np.array(x_train)
x_train = (x_train / 255) - 0.5
x_test = np.array(x_test)
x_test = (x_test / 255) - 0.5


# flatten the images into vectors
x_train = np.array([x.flatten() for x in x_train]).transpose()
x_test = np.array([x.flatten() for x in x_test]).transpose()
# x_train = x_train[:,:1000]


new_images,Up = DigitClassifying.PCA2(x_train, 20)


# plot an example of an image and a reconstructed image
index = 0
img = x_train[:,index]
DigitClassifying.plot_pics(img , index, Up, new_images, y_train)

#run kmean with p=20 and randomize centers
# centers = np.random.random((10, 20)) - 0.5
# clusters,centers,changes = DigitClassifying.Kmeans(10, new_images,
centers)

#test the success using test images
new_images,Up = DigitClassifying.PCA2(x_test, 20)
centers = np.random.random((10, 20)) - 0.5
clusters,centers,changes = DigitClassifying.Kmeans(10, new_images, centers)

#assign each cluster to a digit
clusters_labels,clusters_digits = DigitClassifying.cluster_label(changes,
y_test)
success_rate = DigitClassifying.success_tester(clusters_digits,
clusters_labels)
print("test run success rate is:")
print(success_rate)
```

```
#try the whole process 3 times
new_images,Up = DigitClassifying.PCA2(x_train, 20)
for i in range(3):
    centers = np.random.random((10, 20)) - 0.5
    clusters, centers, changes = DigitClassifying.Kmeans(10, new_images,
centers)
    clusters_labels, clusters_digits =
DigitClassifying.cluster_label(changes, y_train)
    success_rate = DigitClassifying.success_tester(clusters_digits,
clusters_labels)
    print(i,"run success rate is:")
    print(success_rate)

#run kmean with smaller p=12 and compare result

new_images,Up = DigitClassifying.PCA2(x_train, 12)
centers = np.random.random((10, 12)) - 0.5
clusters,centers,changes = DigitClassifying.Kmeans(10, new_images, centers)
clusters_labels, clusters_digits = DigitClassifying.cluster_label(changes,
y_train)
success_rate = DigitClassifying.success_tester(clusters_digits,
clusters_labels)
print("p=12 success rate is:")
print(success_rate)

#run it again but initialize the centers with the mean of 10 images per
label
new_images,Up = DigitClassifying.PCA2(x_train, 20)
centers = np.zeros((10,20))
for center_index in range(10):
    images_found_per_label=0
    k=0
    while images_found_per_label<10:
        if y_train[k]==center_index:
            centers[center_index]+=new_images[k]
            images_found_per_label+=1
        k+=1
    centers[center_index] = centers[center_index]/10

clusters,centers,changes = DigitClassifying.Kmeans(10, new_images, centers)
clusters_labels, clusters_digits = DigitClassifying.cluster_label(changes,
y_train)
success_rate = DigitClassifying.success_tester(clusters_digits,
clusters_labels)
print("centers with the mean of 10 images per label run success rate is:")
print(success_rate)
```

MnistDataloader

```
import inline
import matplotlib
from matplotlib import pyplot as plt
import numpy as np  # linear algebra
import struct
from array import array
from os.path import join
#
# MNIST Data Loader Class
#
class MnistDataloader(object):
    def __init__(self, training_images_filepath,
training_labels_filepath,
                 test_images_filepath, test_labels_filepath):
        self.training_images_filepath = training_images_filepath
        self.training_labels_filepath = training_labels_filepath
        self.test_images_filepath = test_images_filepath
        self.test_labels_filepath = test_labels_filepath

    def read_images_labels(self, images_filepath,
labels_filepath):
        labels = []
        with open(labels_filepath, 'rb') as file:
            magic, size = struct.unpack(">II", file.read(8))
            if magic != 2049:
                raise ValueError('Magic number mismatch, expected
2049, got {}'.format(magic))
            labels = array("B", file.read())

        with open(images_filepath, 'rb') as file:
            magic, size, rows, cols = struct.unpack(">IIII",
file.read(16))
            if magic != 2051:
                raise ValueError('Magic number mismatch, expected
2051, got {}'.format(magic))
            image_data = array("B", file.read())
        images = []
        for i in range(size):
            images.append([0] * rows * cols)
        for i in range(size):
            img = np.array(image_data[i * rows * cols:(i + 1) *
rows * cols])
            img = img.reshape(28, 28)
            images[i][:] = img

        return images, labels

    def load_data(self):
        x_train, y_train =
self.read_images_labels(self.training_images_filepath,
self.training_labels_filepath)
        x_test, y_test =
self.read_images_labels(self.test_images_filepath,
self.test_labels_filepath)
        return (x_train, y_train), (x_test, y_test)
```

DigitClassifying

```
import numpy
from pyparsing.core import Dict
from operator import ne
from numpy.core.fromnumeric import argmin
#
# This is a sample Notebook to demonstrate how to read "MNIST
Dataset"
#
import numpy as np  # linear algebra
import matplotlib.pyplot as plt

def PCA2(pics,p):
    cov = pics@pics.transpose()
    m = len(pics[0])
    cov = (1/m)*cov

    w, v = np.linalg.eig(cov)
    eigenValues , v = w.real , v.real

    x = np.arange(0, len(eigenValues))
    y = np.array(eigenValues)
    # plotting to check if decreasing
    plt.title("eigen values")
    plt.xlabel("X axis")
    plt.ylabel("Y axis")
    plt.plot(x, y, color="green")
    plt.show()

    Up = v[:,0:p]
    Up = Up.transpose()

    # new_images=[]


    new_images = numpy.array([Up@x for x in pics.transpose()])

    return new_images,Up.transpose()

def plot_pics(img , index, Up, new_images, y_train):
    img = img.reshape(28, 28)

    plt.imshow(img, cmap='gray')
    plt.title(y_train[index])
    plt.show()

    rec_img = new_images[index]
    rec_img = Up @ rec_img
    rec_img = rec_img.reshape(28, 28)
    plt.imshow(rec_img, cmap='gray')
    plt.title(y_train[index])
    plt.show()



###########----C----#######
```

```
def Kmeans(k, images, centers):
    # intialize k empty clusters
    clusters = np.empty([k],dtype=object)
    changed = True
    images_centers =np.zeros(len(images)) -2
    # run until clusters doesnt change
    # index = 0
    while (changed):
        # print(index)
        # index+=1
        changed=False
        old_clusters = clusters
        new_clusters = np.empty([k],dtype=object)
        for i in range(k):
            new_clusters[i] = list()
        for i in range(len(images)):
            distances = np.zeros(k)
            for j in range(k):
                # find distance between the image to center
                distances[j] = ((images[i]-centers[j])**2).sum()
            min_index = argmin(distances)
# add the image to the cluster that corresponds to closest center
            new_clusters[min_index].append(images[i])
            if min_index!=images_centers[i]:
                changed = True
                images_centers[i] =min_index
        # update the centers to the mean of every cluster
        for i in range(k):
            if len(new_clusters[i]) > 0:
                sum = 0
                for j in range(len(new_clusters[i])):

                    sum = sum + new_clusters[i][j]
                mean = sum / len(new_clusters[i])
                centers[i] = mean

    return clusters, centers, images_centers

def cluster_label(images_centers, labels):
    cluster_by_digits = np.empty([10],dtype=object)
    for i in range(10):
        cluster_by_digits[i] = list()
    for i in range(len(images_centers)):
        real_val = int(labels[i])

cluster_by_digits[int(images_centers[i])].append(real_val)
    clusters_val = np.zeros(10)
    for i in range(10):
        most_common = find_most_common(cluster_by_digits[i])
        clusters_val[i] = most_common
    return clusters_val,cluster_by_digits

def find_most_common(list):
    counter = np.zeros(10)
    for x in list:
        counter[x]+=1
    return np.argmax(counter)
```

```
def success_tester(cluster_by_digits, clusters_val):
    successes = 0
    counter=0
    for i in range(10):
        for x in (cluster_by_digits[i]):
            counter+=1
            if x == clusters_val[i]:
                successes += 1
    percentage = (successes / counter) * 100
    return percentage
```