Final Exam - CS6220

Nickhil Tekwani - 8/15/2023

readme.txt file attached

Problem 1

KNN Classifier on Noisy Images

DATATSET train_images and test_images. Each row is an image: first column is the label (digit), then the other 784 columns are pixel values.

Task: run a KNN classification algorithm with NN=20 neighbors. You will have to decide the distance function, and other details. Test on the test set and compute accuracy.

These images have noise in them, so clustering with certain distances might not work; to achieve a better result, you need to work on the features first, then rerun your classifier and report accuracy.

In [2]:

```
import numpy as np
from sklearn.decomposition import PCA

def load_dataset(filename):
    data = np.loadtxt(filename)
    labels = data[:, 0].astype(int)
    images = data[:, 1:]
    return labels, images

train_labels, train_images = load_dataset('trainPB1.txt')
test_labels, test_images = load_dataset('testPB1.txt')

# preprocess the images using PCA
pca = PCA(n_components=40) # retain 40 components
pca_train_images = pca.fit_transform(train_images)
pca_test_images = pca.transform(test_images)

# cosine similarity function
def cosine_similarity(v1, v2):
    return np.dot(v1, v2) / (np.linalg.norm(v1) * np.linalg.norm(v2))

# KNN classifier implementation
def knn_classifier(train_images, train_labels, test_image, k=20):
    distances = []
    for i in range(len(train_images)):
        sim = cosine_similarity(test_image, train_images[i])
        distances.append((train_labels[i], sim))
    sorted_distances = sorted(distances, key=lambda x: x[1], reverse=True)
    top_k = sorted_distances[:k]

    # use majority voting
    votes = {}
    for label, _ in top_k:
        if label in votes:
            votes[label] += 1
        else:
            votes[label] = 1

    return max(votes, key=votes.get)

# finally, test on the test set and compute accuracy
correct = 0
for i in range(len(pca_test_images)):
    predicted_label = knn_classifier(pca_train_images, train_labels, pca_te
    if predicted_label == test_labels[i]:
        correct += 1

accuracy = correct / len(pca_test_images) * 100
print(f'Accuracy: {accuracy}%')
```

```
Accuracy: 78.55%
```

Problem 2

Quiz Difficulties Mixture

Professor Virgil teaches intro-math CS1800 to 1500 students. For the quiz there are two versions A,B; each version has difficulties that can result in three grade outcomes ["fail=0" "pass=1" excellent=2"] with respective probabilities [q0_A, q1_A, q2_A].sum=1 for quiz version A and [q0_B, q1_B, q2_B].sum=1 for quiz version B (outcomes are independent for each student, as they are equally good at math). Virgil splits the students into 50 sessions of S=30 students each; each session gets a quiz with fixed non-uniform selection probabilities pi_A, pi_B (which sum to 1). Attached is the gradesheet , each row corresponds to a session with 30 ternary grades for students in that session.

Compute the probabilities to assign each test to session ("pi" probabilities), and also the chance of students at large solving each problem (6 "q" probabilities). HINT: for each session, since students are independent of each other, what matters is the number of students x solving the problem out of the session size S.

In [10]:

```
import numpy as np

def has_converged(old_vals, new_vals, threshold=1e-6):
    old_vals_flattened = [item for sublist in old_vals for item in sublist]
    new_vals_flattened = [item for sublist in new_vals for item in sublist]
    return np.linalg.norm(np.array(old_vals_flattened) - np.array(new_vals_

# initialize parameters
q_A = np.array([.595, .27, .155])
q_B = np.array([.30, .38, .32])
pi = np.array([0.74, 0.26])

old_q_A, old_q_B, old_pi = np.zeros(3), np.zeros(3), np.zeros(2)

def e_step(grades, q_A, q_B, pi):
    # calculate expected counts given current parameters for each session
    expected_counts_A = []
    expected_counts_B = []

    for session in grades:
        likelihood_A = np.prod([q_A[grade] for grade in session])
        likelihood_B = np.prod([q_B[grade] for grade in session])

        prob_A = likelihood_A * pi[0] / (likelihood_A * pi[0] + likelihood_
        prob_B = 1 - prob_A

        expected_counts_A.append(prob_A)
        expected_counts_B.append(prob_B)

    return expected_counts_A, expected_counts_B

# multinomial distribution
def m_step(grades, expected_counts_A, expected_counts_B):
    total_students = len(grades) * len(grades[0])
    total_A = sum(expected_counts_A)
    total_B = sum(expected_counts_B)

    total_sessions = len(grades)

    q_A = np.zeros(3)
    q_B = np.zeros(3)

    for i in range(3):  # for each grade (0, 1, 2)
        for session, count_A, count_B in zip(grades, expected_counts_A, exp
            q_A[i] += session.count(i) * count_A
            q_B[i] += session.count(i) * count_B

    # normalize the q values to ensure they represent probabilities
    q_A /= q_A.sum()
    q_B /= q_B.sum()

    pi_A = total_A / total_sessions
    pi_B = total_B / total_sessions

    # normalize the pi values so they sum to 1
    total_pi = pi_A + pi_B
    pi_A /= total_pi
```

```
    pi_B /= total_pi

    return q_A, q_B, [pi_A, pi_B]


# load grade data
grades = []
with open('studentgrades_pb2.txt', 'r') as f:
    for line in f:
        grades.append(list(map(int, line.strip().split())))

# EM algo with convergence criteria
iteration = 0
while not (has_converged([old_q_A, old_q_B, old_pi], [q_A, q_B, pi])):
    old_q_A, old_q_B, old_pi = q_A.copy(), q_B.copy(), pi.copy()

    expected_counts_A, expected_counts_B = e_step(grades, q_A, q_B, pi)
    q_A, q_B, pi = m_step(grades, expected_counts_A, expected_counts_B)

    iteration += 1
    if iteration > 1000:  # max iteration cap to ensure it doesn't run inde
        break

r = 3
print(f"pi: [{round(pi[0], r)}, {round(pi[1], r)}]")
print(f"q_A: [{round(q_A[0], r)}, {round(q_A[1], r)}, {round(q_A[2], r)}]")
print(f"q_B: [{round(q_B[0], r)}, {round(q_B[1], r)}, {round(q_B[2], r)}]")
```

```
pi: [0.632, 0.368]
q_A: [0.641, 0.212, 0.148]
q_B: [0.302, 0.415, 0.282]
```